UNCUYO
UNIVERSIDAD NACIONAL DE CUYO

FACULTAD DE INGENIERÍA

Inteligencia Artificial II

U1: Búsqueda y Optimización

Grupo N°2

Alumno:	**Legajo:**
Cantú, Maximiliano	11294
Costarelli, Agustín	10966
Lage Tejo, Joaquín	11495

2 0 2 0

Trabajo Práctico Nº 1
Búsqueda y Optimización

Ejercicio 1 - Problemas:

a) Diseño de un proceso de manufactura

Se utilizaría un Problema de Satisfacción de Restricción (en adelante CSP) con un Temple Simulado como algoritmo auxiliar, conformando una búsqueda local. Esto se debe a las restricciones de la obra (asumiendo que ya ha sido diseñada, hay ciertas tareas a realizar en un orden específico, pero tomarlas como un "todo" nos permite alterar su orden respecto a otro paquete), pero además la necesidad de disminuir el espacio de búsqueda significativamente para reducir la complejidad del problema.

La función de Temperatura decreciente se considera en forma lineal.

Estado Inicial: Parto de una estado completo (todas las variables son asignadas) pero probablemente inconsistente (se violan restricciones), generado aleatoriamente. En este caso serían el orden procesos de manufactura, o las máquinas a emplear.

Modelo de transición: Se evalúa la calidad del estado actual y revisan las restricciones, se elige si movernos a vecinos o no (estados dónde varía el valor de solo una variable), para lo cual se tiene en cuenta su heurística, si es peor se elige o no con una probabilidad que va decreciendo.

Heurística: Elegimos la variable más conflictiva, es decir aquella que participa en el mayor número de restricciones.

Condición de parada: Cuando t=0

b) Planificación de órdenes de fabricación

Podría plantearse un Temple Simulado con permutaciones, asumiendo que el orden de los productos no es importante y que la única restricción es el uso de las distintas máquinas para distintos productos (descartando un CSP), aunque también puede plantearse un Algoritmo Genético. Como la evaluación, asumimos, sería solo del tiempo de producción y el uso de cada máquina, no sería necesario implementar un algoritmo de búsqueda auxiliar.

Como modelo, suponemos que existen N máquinas que realizan todos los productos, y M distintos tipos de productos con su tiempo específico. Existe una matriz de N filas con M columnas. El objetivo es obtener la combinación que reduzca el tiempo de producción al mínimo. La complejidad del problema sería entonces de (N*M!).

La función de Temperatura decreciente se considera en forma exponencial

Estado Inicial: Genero el estado de forma aleatoria ubicando todos los productos necesarios, distribuidos en ambas filas.

Modelo de transición: Evalúa el estado actual, y calculo sus estados vecinos (realizando 1 sola permutación entre todas las máquinas). Evaluó el estado vecino y lo elijo o no considerando si es mejor, o si es peor, con una probabilidad que va decreciendo.

Función de calidad: Se calcula el tiempo de producción de cada máquina, y se toma el de todo el proceso como el máximo.

Condición de parada: Cuando t=0

c) Encontrar la ubicación óptima de aerogeneradores en un parque eólico

Algoritmo Genético, dado que el problema podría plantear múltiples posiciones posibles con una cantidad fija de aerogeneradores a colocar (posiblemente menor que la cantidad deposiciones). Los individuos (binarios) estarían compuestos por las posiciones indicando si existe un aerogenerador (1) o no (0) en cada posición. Este sería un modelo simplificado de un caso real.

Estado Inicial: Población de individuos binarios creado aleatoriamente, con la restricción dura de tener que poseer exactamente el número de aerogeneradores a colocar.

Modelo de transición: Se evalúa su fitness (función de calidad), se elige un porcentaje acotado (por ejemplo 30%) de la población como padres de la próxima. Se realiza un crossover y luego se mutan los genes (mutando tantos como sea necesario para obtener la cantidad de aerogeneradores exacta cada individuo).

Función de calidad: Se conoce de antemano la producción prevista en cada posición además de su costo y cómo interfiere con las demás. Esto se evalúa en cada individuo para obtener su valor de “fitness”.

Condición de parada: Una condición doble, si los mejores de dos generaciones consecutivas son iguales y sus promedios difieren en una cantidad menor a la estipulada (una cierta tolerancia), se acepta al mejor de la nueva generación como resultado final.

d) Planificar trayectorias de un brazo robotizado con 6 grados de libertad

Algoritmo A*, debido a que el problema requiere el movimiento más óptimo para llegar de un punto al otro, con lo cual deberíamos evaluar la mayor cantidad de opciones posibles. Al

mismo tiempo, la complejidad no sería alta ya que está definida por el alcance del brazo, que no varía luego de su construcción. La existencia de obstáculos puede, inclusive, disminuir el espacio de búsqueda y acelerar el proceso. La complejidad es de 8

Estado Inicial: Posición actual del brazo.

Modelo de transición: Generar los hijos (posiciones vecinas) y evaluar si es un estado posible, junto con su heurística, a fin de determinar cuál de los hijos expandir en la siguiente iteración.

Función heurística: Asumimos que el brazo puede moverse en forma diagonal, por lo tanto la distancia en línea recta (euclidiana) es la mejor heurística posible, ya que es admisible, porque no sobreestima la distancia (en el peor caso solo la subestima). Esto asegura un algoritmo completo y óptimo.

Condición de parada: Solo se detiene si llegó al punto final (si lo elige como nuevo punto a expandir, chequea si se encuentra en el punto final). O dado el caso de que no sea posible encontrar un camino debido a obstáculos.

e) Diseño de un generador

Se utilizaría un Problema de Satisfacción de Restricción (en adelante CSP) con un AG como algoritmo auxiliar, conformando una búsqueda local. Esto se debe a las restricciones necesarias dadas las características que debe tener el generador, pero además la necesidad de disminuir el espacio de búsqueda significativamente para reducir la complejidad del problema. A medida que aumento las restricciones, la influencia del AG se vuelve menor.

Estado Inicial: Parto de una población (AG) con individuos que sean una solución completa (todas las variables tienen un valor) pero probablemente inconsistente, generada aleatoriamente.

Modelo de transición: Se evalúa su fitness (función de calidad) al mismo tiempo que se revisan las restricciones, se elige un porcentaje acotado (por ejemplo 30%) de la población, en función de la violación de la menor cantidad de restricciones y su fitness; como padres de la próxima. Se realiza un crossover y luego se mutan los genes (podemos definir que se muten solo aquellos correspondientes a la variable más o menos restringida, por ejemplo), con lo cual se altera el valor da alguna de las variables del CSP.

Heurística: Elegimos la variables más conflictiva, es decir aquella que participa en el mayor número de restricciones.

Función de calidad: Suponiendo que las restricciones son las características funcionales del generador, su función de calidad podría ser el precio de construcción (en función de los métodos elegidos o los materiales). Con ello el problema se transforma en uno de minimización de costos.

Condición de parada: Se elige aquella solución que no viola ninguna restricción, y que posee el mejor fitness, dentro de un período de tiempo o si los resultados se repiten en generaciones consecutivas.

f) Definición de una secuencia de ensamblado óptima

Usamos un CSP con un A* como algoritmo auxiliar, conformando una búsqueda global. Esto se debe a las restricciones necesarias dadas la secuencia a seguir (qué paso sigue necesariamente a otro), pero además la necesidad de optimizar el problema nos fuerza a utilizar un método de búsqueda global, y sabemos que el A* es un buen ejemplo.

Estado Inicial: Parto de un estado que asigna un valor alguna de las variables (heurística).

Modelo de transición: Expando el nodo asignando un valor a otra variable hasta ahora no asignada, evalúo si no infringe ninguna restricción, y luego se elige el nodo a expandir de acuerdo a la función de calidad y la heurística planteada.

Heurística: Elegimos la variables restringida, es decir aquella que puede adoptar la menor cantidad de valores.

Función de calidad: tiempo de ensamblado total.

Condición de parada: Se conoce que el primer resultado encontrado por A* es el óptimo, por lo tanto el programa termina al encontrar una solución que asigna valores a todas las variables, y cumple con todas las restricciones.

g) Planificación del proyecto de una obra

Se utilizaría un Problema de Satisfacción de Restricción (en adelante CSP) con un Temple Simulado como algoritmo auxiliar, conformando una búsqueda local. Esto se debe a las restricciones de la obra (asumiendo que ya ha sido diseñada, hay ciertas tareas a realizar en un orden específico, pero tomarlas como un “todo” nos permite alterar su orden respecto a otro paquete), pero además la necesidad de disminuir el espacio de búsqueda significativamente para reducir la complejidad del problema.

La función de Temperatura decreciente se considera en forma lineal.

Estado Inicial: Parto de una estado completo pero probablemente inconsistente, generado aleatoriamente. En este caso serían el orden de las tareas a realizar.

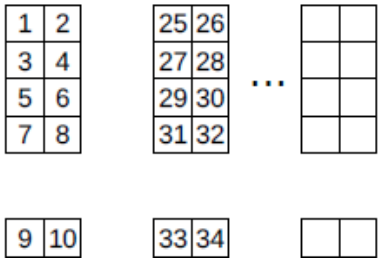
Modelo de transición: Se evalúa la calidad del estado actual y revisan las restricciones, se elige si movernos a vecinos o no (estados dónde varía el valor de solo una variable), para lo cual se tiene en cuenta su heurística, si es peor se elige o no con una probabilidad que va decreciendo.

Heurística: Elegimos la variable más conflictiva, es decir aquella que participa en el mayor número de restricciones.

Condición de parada: Cuando t=0

Ejercicio 3

Se analiza primero este ejercicio dado que fue el desarrollado en primer lugar (antes que el 2), por su menor complejidad.



Sabemos que los algoritmos A* son completos (si hay una solución la encuentra) y óptimo (la primera solución encontrada es la mejor), si su heurística es admisible. Esto significa principalmente que no debe sobreestimar el costo de llegar al objetivo. La heurística utilizada es la distancia euclidiana (distancia en línea recta) desde el punto de inicio al punto final (o meta, ambos aleatorios). El espacio de búsqueda sigue la distribución mostrada en la imagen.

Es un algoritmo de A* que utiliza principalmente dos clases (clase Nodo y clase A*). La clase Nodo se encarga de evaluar los atributos de cada nodo (en este caso, su costo desde el punto de inicio g(n), su heurística h(n) y su función de evaluación f(n)). La clase A* es la que posee los métodos de resolución iterativa del problema.

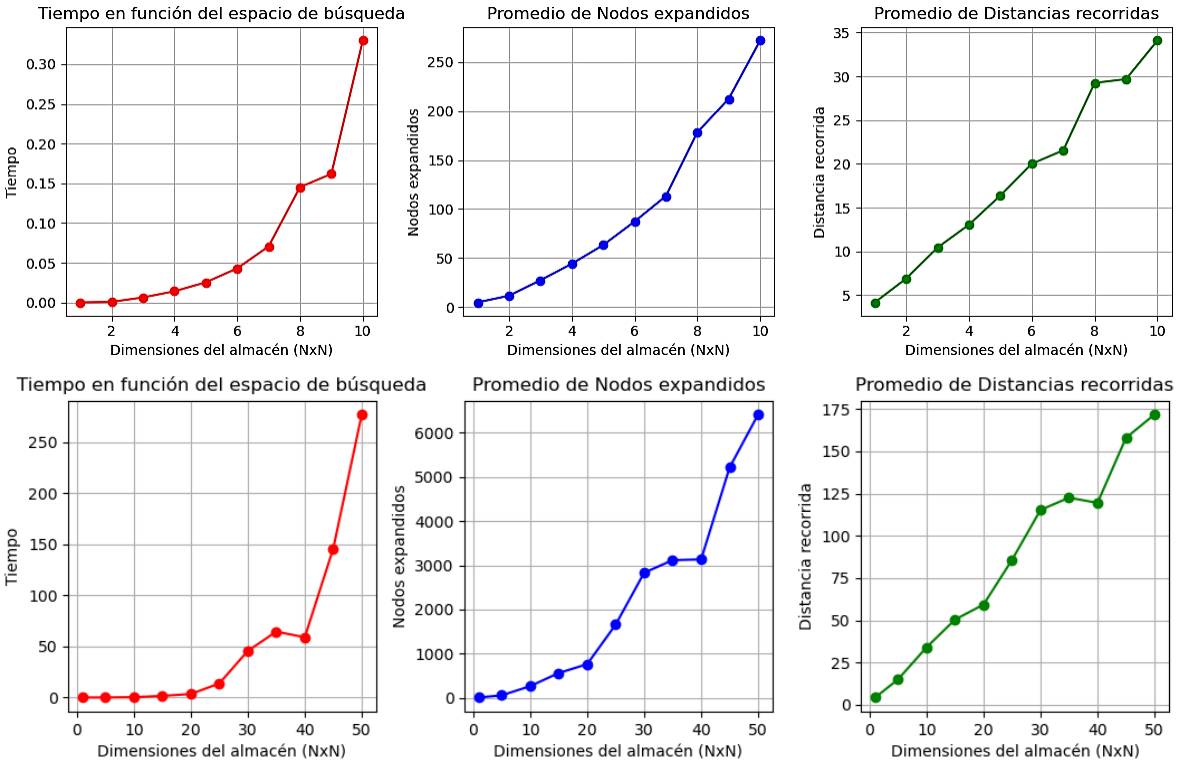
La clase A* trabaja con 3 listas: Lista Abierta (nodos por expandir), Lista Bloqueada (nodos que no pueden ser expandidos, en este caso los ocupados por las estanterías), y Lista Cerrada (nodos ya expandidos).

El proceso consiste en tomar el nodo con menor f(n) (en la primera iteración es el INICIO) y expandirlo (encontrar sus vecinos). Dicho nodo pasa a estar en la lista cerrada, y luego de corroborar que sus vecinos son nodos válidos (fuera de Lista B, función “repetidos”; y dentro del espacio de búsqueda) se agregan a la Lista A como posibles nodos a expandir. Luego se

selecciona de dicha lista aquel con el menor f(n), pero antes de expandirlo se verifica que no sea el nodo meta (para el h(n)=0), si es el caso, el problema se termina.

La función f(n) se calcula para cada nodo, y sabemos que g(n) aumenta en 1 a medida que nos al jamos del inicio, y h(n) va decreciendo (nos acercamos a la meta). Para esto se emplean varios métodos dentro de la clase A*, que realizan tareas de verificación, creación de vecinos, evaluación de los nodos, etc.

Para evaluar el desempeño del algoritmo, se produjeron los puntos inicial y final al azar, y se registró el tiempo promedio de resolución, la distancia promedio de la solución y los nodos expandidos en promedio, para llegar a la solución. Se realizaron 100 repeticiones en los tamaños de espacio de 1x1 a 10x10; y 50 en los espacios de 1x1 a 50x50 (debido al tiempo de procesamiento).



Se observa claramente como las tendencias en las curvas se mantienen en ambos casos. La distancia promedio que debe recorrer crece de forma casi lineal, lo que es lógico ya que esta varía con el aumento de N (filas y columnas de estanterías, de 8 lugares cada una). El promedio de Nodos Expandidos crece en forma de función potencial (casi de forma cuadrática), lo cual guarda relación con el aumento cuadrático del espacio de búsqueda (NxN). Finalmente el tiempo de búsqueda crece de forma exponencial también, pero con una pendiente aún mayor (algo mayor a una cúbica).

Debemos recordar que este algoritmo siempre encuentra la solución, ya que siempre existe (la cantidad de estanterías es proporcional al tamaño del almacén y no es posible que el camino esté bloqueado); la única variación es el tiempo (o los nodos a expandir) que le llevará, creciente de acuerdo a alguna función potencial. Esto es consistente con la complejidad de un algoritmo A* que es $O(b^d)$, siendo en nuestro caso b(branching factor)=3 en promedio y d(depth) la distancia máxima entre inicio y fin (en el peor de los casos se hallarán en esquinas opuestas, siendo así N+N).

Ejercicio 2

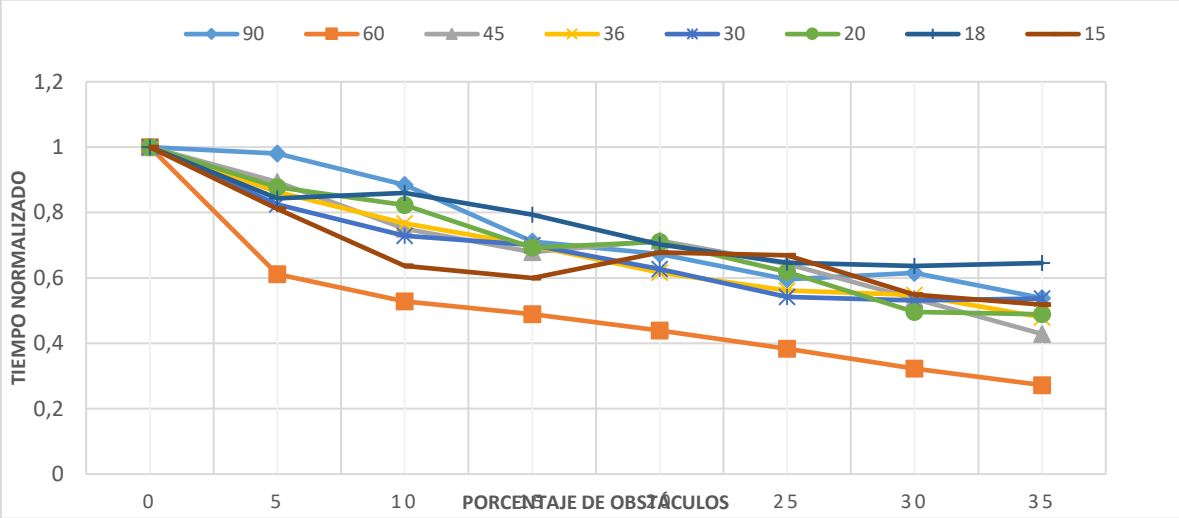
En este caso se trata de encontrar el camino de un brazo robótico en el espacio. Se asume que el brazo solo puede mover una articulación a la vez. Su complejidad es de 6^d, siendo "d" la distancia máxima que puede llegar a recorrer.

Paso (Grados)	Nodos	Sin Obstáculos
180	8	5
90	27	24
60	64	61
45	125	122
36	216	213
30	343	340
20	1000	997
18	1331	1328
15	2197	2194
12	4096	4093
10	6859	6856
5	50653	50650

Este ejercicio sigue el mismo razonamiento que el anterior, ya que es resuelto de igual forma con A*. Las diferencias fundamentales consisten en que el espacio de búsqueda es ahora tridimensional, y se suma la aleatoriedad de los obstáculos en el mismo. Esto agrega complejidad al problema, y la opción de encontrarnos con un problema sin solución (un camino totalmente bloqueado entre Inicio y Meta). Este caso sucede a nivel algoritmo cuando se vacía la Lista Abierta, no quedan nodos por expandir.

En la tabla anexa se pueden observar la cantidad de nodos totales existentes en el espacio, de acuerdo al grado de avance de las 3 articulaciones. Sabemos que toma valores graduales entre 0 y 360 (siendo este el mismo). Si el paso es de 180 (paso máximo), cada articulación podrá tener solo 2 posiciones (0 o 180), de modo que $2^3=8$. En el caso de evaluar el código sin obstáculos (porcentaje 0%), la cantidad de nodos que se expanden son siempre el total menos 3. Es decir que, no se expande ni el nodo meta (como explicamos en el ejercicio 2) ni otros dos nodos que quedan siempre en posiciones inmediatamente desfavorables luego de elegir el siguiente nodo a expandir luego del Inicio.

Si bien el problema corre normalmente con aleatoriedad en obstáculos tanto como en Inicio y Fin, para evaluar su desempeño, se fijaron el Inicio y Meta en puntos lo más alejados posibles (esquinas opuestas de un cubo), con la finalidad de evaluar solamente la influencia del paso elegido, y del porcentaje de obstáculos existentes.

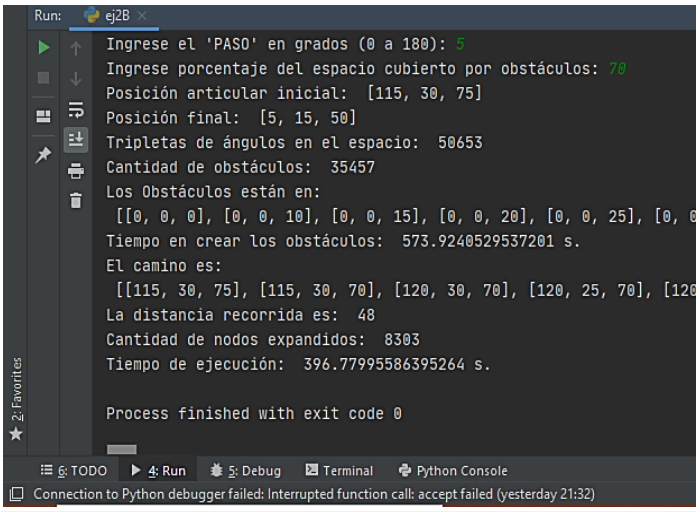


Recordemos que a mayor paso, menor cantidad de nodos. Lo que se hizo fue recopilar los tiempos promedio de resolución con distintos valores de paso y porcentaje, y luego se los normalizo respecto al tiempo sin obstáculos (0%, correspondiente a 1), a modo de poder evaluar todos los casos en un mismo gráfico. Es interesante observar como la existencia de obstáculos disminuye la expansión de nodos, haciendo que el algoritmo encuentre más rápidamente el camino (si es que existe). Vemos que hay una marcada tendencia a disminuir el tiempo a mayor porcentaje. En otras palabras, se poda el árbol de búsqueda, disminuyendo la complejidad del problema.

Pruebas con mayor porcentaje de obstáculos no pudieron ser realizadas por dos motivos: aumentar el porcentaje también aumenta la posibilidad de no tener un camino posible, lo que paraba el código e impedía seguir buscando; y el tiempo de cómputo total crecía notablemente. Con esto nos referimos al tiempo

Paso de 5° (50653 nodos)		
Porcentaje	**Obstáculos**	**Tiempo(s)**
10	5065	9.34
50	25326	286.96
70	35457	573.92
90	45587	900.48
99	50146	2754.45

empleado tanto en crear los obstáculos como en encontrar la solución. Se muestran ejemplos para un paso de 5° con distintos porcentajes, y una gráfica evaluando su influencia.



Vemos (en el ejemplo mostrado) como el tiempo de resolución es inclusive menor que el empleado para la creación de los obstáculos, que no es el tema de estudio al que nos estamos enfocando primordialmente.

En la gráfica vemos el aumento en forma exponencial del tiempo empleado. Esto no influiría en una situación real dónde los obstáculos están dados por el entorno. Por esta razón se desistió en utilizar esta función de aleatoriedad en los problemas siguientes.

Ejercicio 5

Para el uso de temple simulado se procedió a darle uso nuevamente al algoritmo de A* junto con la distribución del depósito de los ejercicios anteriores. Cabe aclarar una leve diferencia: en el ejercicio 3 las posiciones se evaluaron considerando un espacio matricial, es decir (filas, columnas); mientras que en el ejercicio 5 (y en el 6) se consideraron posiciones en un plano (x, y). Esto solo influye en la lectura y comprensión de los datos, sin esta consideración, un espacio de búsqueda es la matriz transpuesta del otro.

Los parámetros del algoritmo son Temperatura inicial, Temperatura final, los objetos a analizar y el factor de enfriamiento.

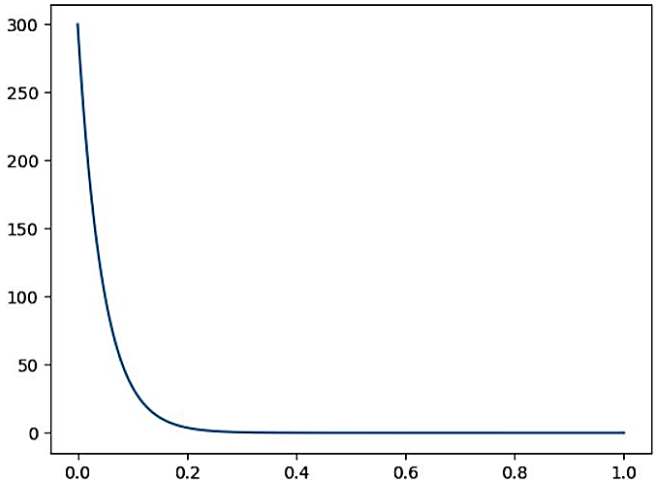
Como medida de enfriamiento se decidió tomar el valor de la temperatura y multiplicarlo por el factor de enfriamiento. Es decir que este factor siempre tiene que ser menor a 1.

Dada la facilidad de la programación en clases se facilitó el código. Se encuentra la clases A* del ejercicio anterior (3). El algoritmo de temple consta de una clase Temple, esta recibe como parámetros de entrada los valores ya mencionados. A su vez la clase temple hace uso de la clase recolector que representa al robot que tendría que recolectar los objetos en el orden dado y este mismo nos informaría cual ha sido el costo del camino, cada cuadro del mapa vale 1 a fines del costo. El algoritmo para cada iteración realiza una permutación simple de la lista de objetos y se calcula el costo de recolectarlos en dicho orden, se compara ese valor con uno previamente guardado, si el nuevo costo es menor pasa a ser el orden de objetos guardados, si el costo no es menor se hace por probabilidad la selección. Dentro de la probabilidad se incluye el valor de T, y a mayor T son mayores las probabilidades de elegir un valor peor.

En esta grafica se puede observar la curva de temperatura y como se enfría para dar una idea como varia la probabilidad de elegir peores valores a medida que el algoritmo se ejecuta.

Para nuestro algoritmo se introdujeron los siguientes valores:

- Temperatura Inicial = 3000
- Temperatura Final = 0.0000001
- Costo Enfriamiento = 0.8

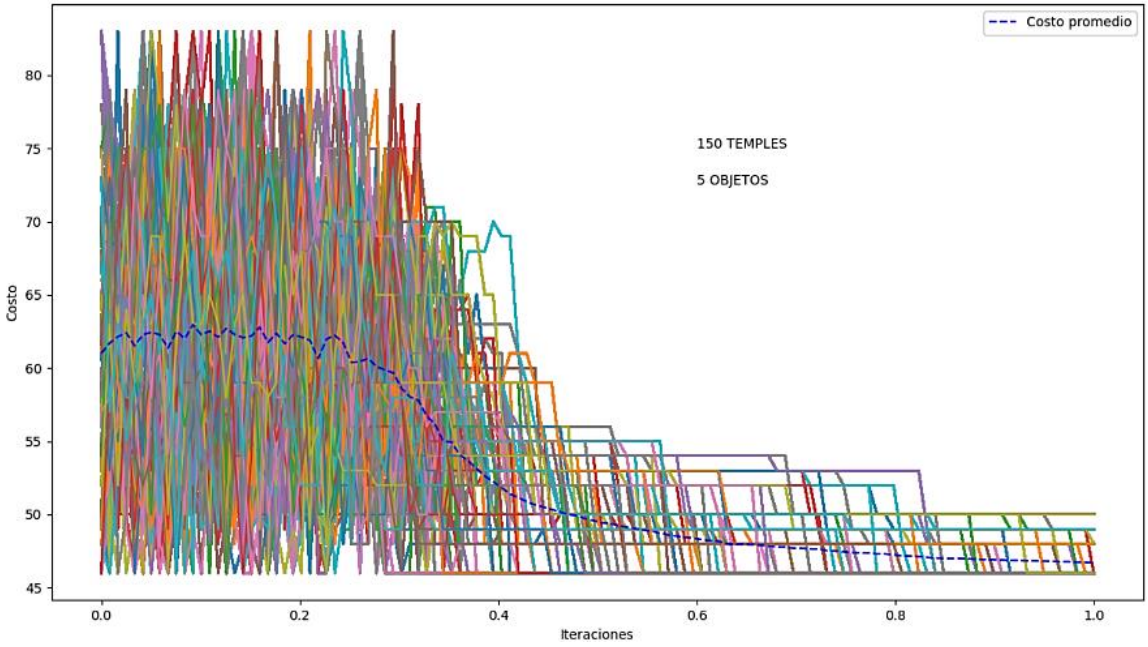


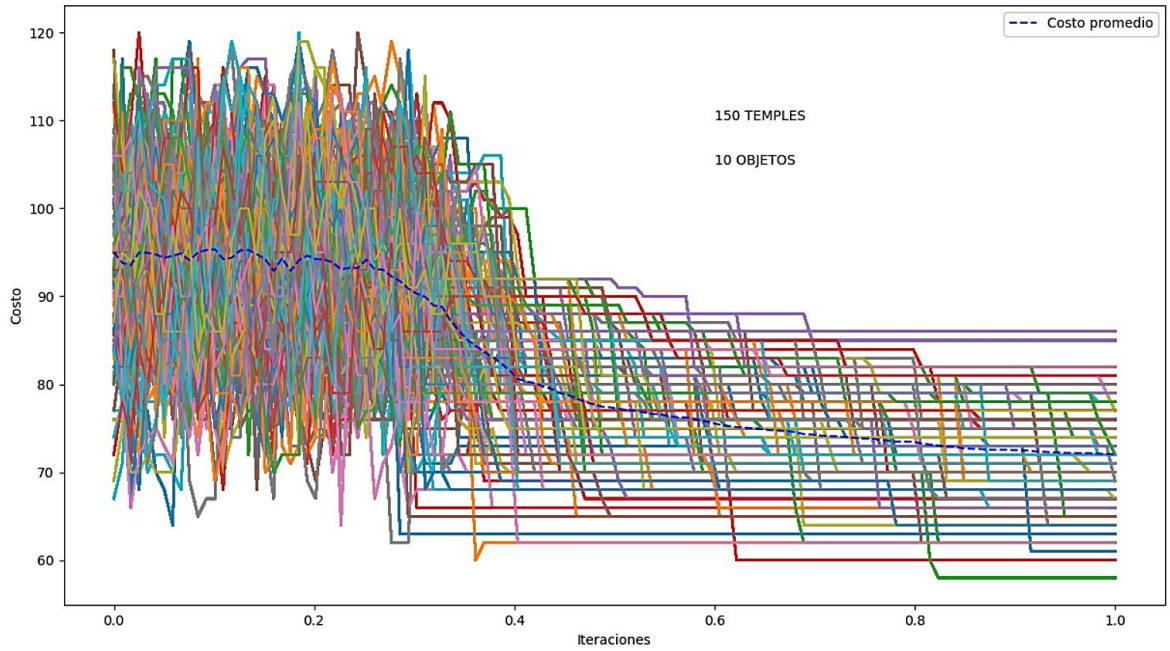
Obtenidos a prueba y error, y manteniendo la cantidad de objetos e iteraciones fijas, hasta observar los mejores resultados.

Luego de obtener valores de temperatura y enfriamiento óptimos, ejecutamos el algoritmo múltiples veces, variando la cantidad de objetos ingresados y realizando 150 iteraciones para cada caso, con el fin de validar dichos valores y sabiendo que el temple simulado es un algoritmo de búsqueda local.

A continuación, se detallan los objetos ingresados en cada caso y las respuestas obtenidas gráficamente. Además, graficamos de manera superpuesta, el costo promedio de las 150 iteraciones, y así poder observar la respuesta promedio del algoritmo:

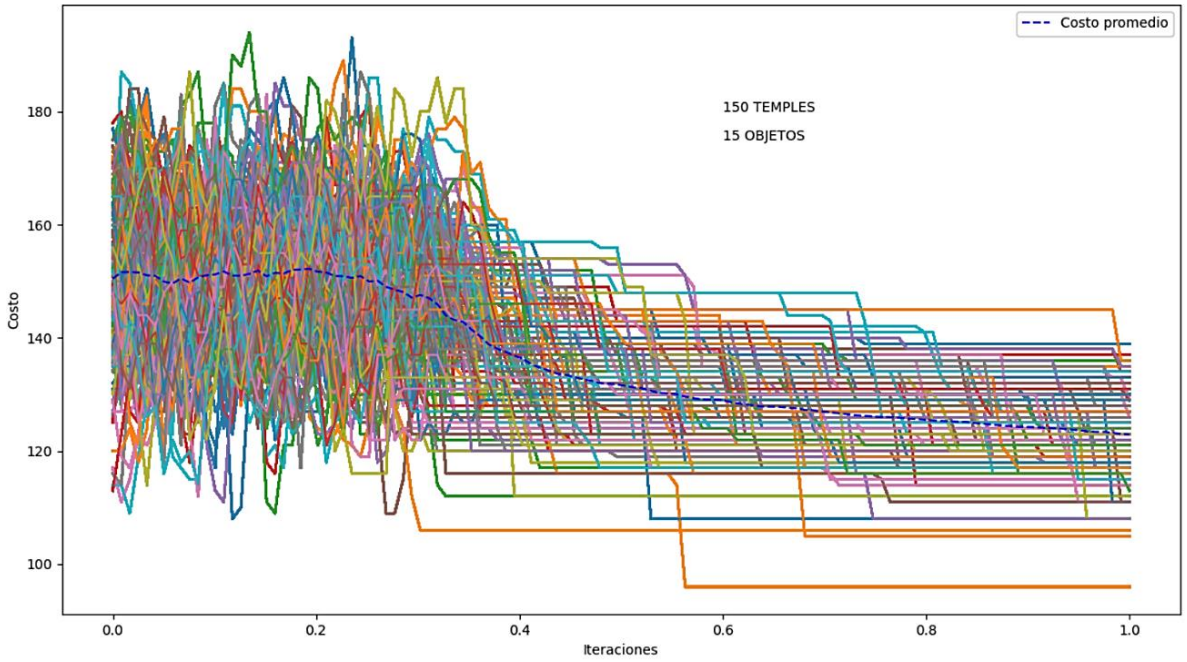
- 5 Objetos = [7, 48, 28, 31, 15]





- 10 Objetos = [7, 48, 28, 31, 15, 43, 23, 13, 9, 27]

- 15 Objetos = [7, 48, 28, 31, 15, 43, 23, 13, 9, 27, 37, 18, 10, 35, 5]



Podemos observar que, en cada caso, el algoritmo tiende a converger a un valor de costo promedio, y así podemos asegurar que el algoritmo es capaz de resolver correctamente el problema.

Ejercicio 6

Para realizar este ejercicio, se reutilizaron los algoritmos A* y Temple Simulado, con sus mismos parámetros de inicio, mediante la técnica mencionada anteriormente de programación en clases.

Inicialmente al algoritmo se le ingresan como parámetros iniciales el tamaño de la población, la cantidad de órdenes a realizar, la cantidad de objetos en cada orden, donde los objetos se generan de manera aleatoria, y una lista de 2 valores que corresponden a las filas y columnas de las estanterías, con los cuales se genera el entorno de trabajo.

De forma general, el algoritmo procede a utilizar la clase A* para ubicar el camino más corto hasta el objetivo, luego utiliza el de temple simulado para para optimizar la búsqueda de todos los productos y finalmente mediante el algoritmo genético, logar optimizar la ubicación de los productos en los estantes del almacén en base a la forma óptima de buscarlos.

Al finalizar obtenemos una población con la misma cantidad de individuos que al inicio, pero optimizada en función al fitness de cada uno, y un valor de fitness para cada individuo. Como último paso, se procede a la selección del menor valor de fitness, ya que simboliza el menor costo, para así poder seleccionar el mejor individuo de dicha población y obtener el resultado

final del ejercicio. Dicho resultado representa la óptima disposición de los productos en los estantes, para que un recolector pueda buscar cada producto de manera eficiente y siguiendo el camino más corto, siempre considerando que la posición final del recolector es la misma que la inicial.

Se realizó la ejecución del algoritmo para los siguientes valores de entrada:

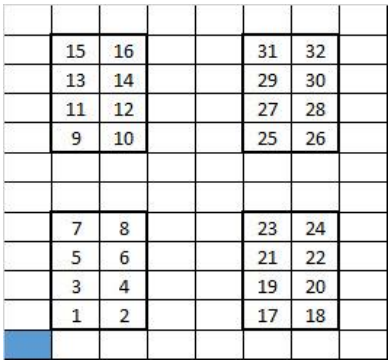
- Tamaño de la población = 4
- Cantidad de órdenes = 4
- Cantidad de objetos por orden = 3
- Disposición de estanterías = [2,2]

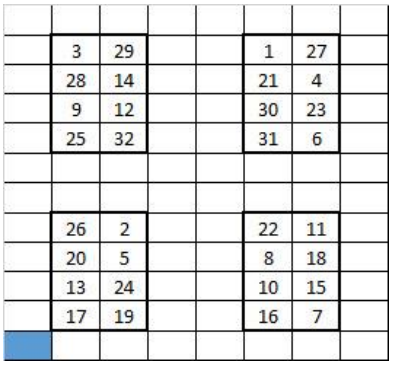
Y se obtuvo como resultado que la mejor solución es:

[17, 19, 13, 24, 20, 5, 26, 2, 25, 32, 9, 12, 28, 14, 3, 29, 16, 7, 10, 15, 8, 18, 22, 11, 31, 6, 30, 23, 21, 4, 1, 27]

Con un valor de fitness de: 69

Aclaramos que el almacén con el que trabajamos es de 4 estanterías dispuestas en 2 filas y 2 columnas, en las cuales hay 8 productos en cada una. Por lo tanto, el primer grupo de 8 valores de la solución corresponden a la primera estantería ubicada en fila 1 y columna 1, tomando como referencia la esquina inferior izquierda del mapa; el segundo grupo corresponde a la estantería ubicada en la fila 2 y columna 1; el tercer grupo a la ubicada en la fila 1 y columna 2 y, por último, el cuarto grupo de 8 valores corresponde a la estantería ubicada en la fila 2 y columna 2. En las siguientes imágenes podemos observar un plano inicial (Fig. 1) de las estanterías del almacén y un plano final (Fig. 2) de acuerdo con lo q se explicó anteriormente. Cabe destacar que el cuadro resaltado en azul en ambas figuras representa la posición inicial y final del recolector, la cual es la casilla [0,0].





Ejercicio 7

Para la implementación se tuvieron en cuenta las siguientes consideraciones. Se eligieron una vez de manera aleatoria y fijada para el resto del problema las máquinas y sus tipos. Se contaba con 4 máquinas, maquina 1 tipo A, maquina 2 tipos A, B y C, maquina 3 tipos C y D y maquina 4 tipos D. Cada una solo puede procesar una tarea a la vez.

Como algoritmo se utilizó uno de búsqueda local del tipo greedy. Con este no guardábamos información sobre estados anteriores, al ser de búsqueda local no podíamos asegurar que la respuesta fuera la óptima. Además de que nos quedábamos con el primer resultado obtenido.

En nuestro problema encontramos:

Estado inicial:

- Lista de tareas ordenadas, con su respectivo tipo y duración
- Maquinas con su respectivo tipo y si está disponible.

Estados:

- Se revisaba para cada máquina disponible cual era la primera tarea asignable para el periodo actual.

Acción:

- Se asignaba la tarea y se actualizaba el estado de la máquina.

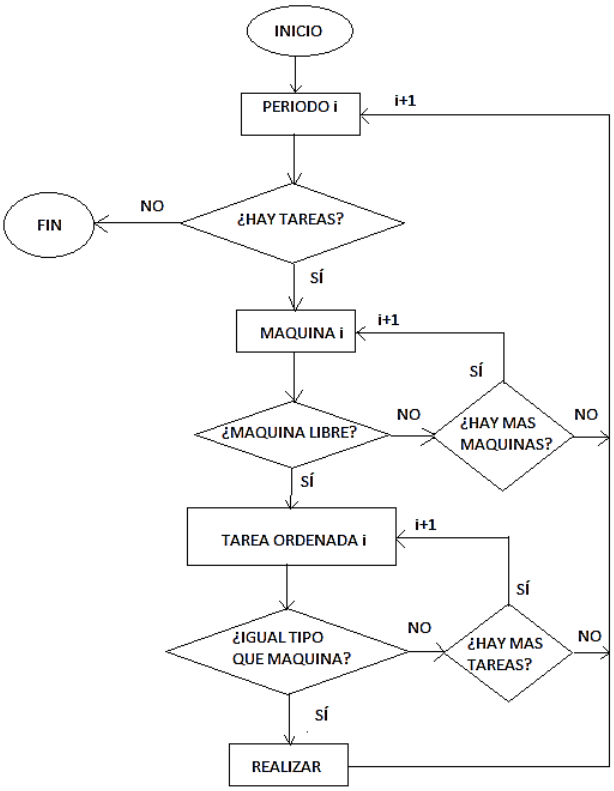
Prueba de meta:

- Todas las maquinas disponibles tienen una tarea del tipo correspondiente asignado en un periodo correcto.

El criterio para resolver el problema fue ordenar las tareas por realizar de acuerdo a su duración, dando prioridad a las de menor duración, luego se revisó de qué tipo de tarea había más y se ponían adelante de la lista ordenada. De esa manera uno se podía asegurar de ir atendiendo los tipos de tarea por igual.

Los parámetros importantes para nuestro problema son 3:

Periodo, es un valor que comienza en 0 y se va aumentando de a 1 para ver la línea del tiempo del problema. La lista de tareas por realizar con su respectiva duración. Las maquinas, que tenían indicado el periodo en el cual se liberaban, todas por defecto comienzan en 0.



El proceso consiste en elegir una maquina disponible. Luego en base a la lista de tareas ordenadas, bajo nuestro criterio, se revisa cual es la primera tarea que cumple con la restricción del tipo de esa máquina y se inicia la tarea.

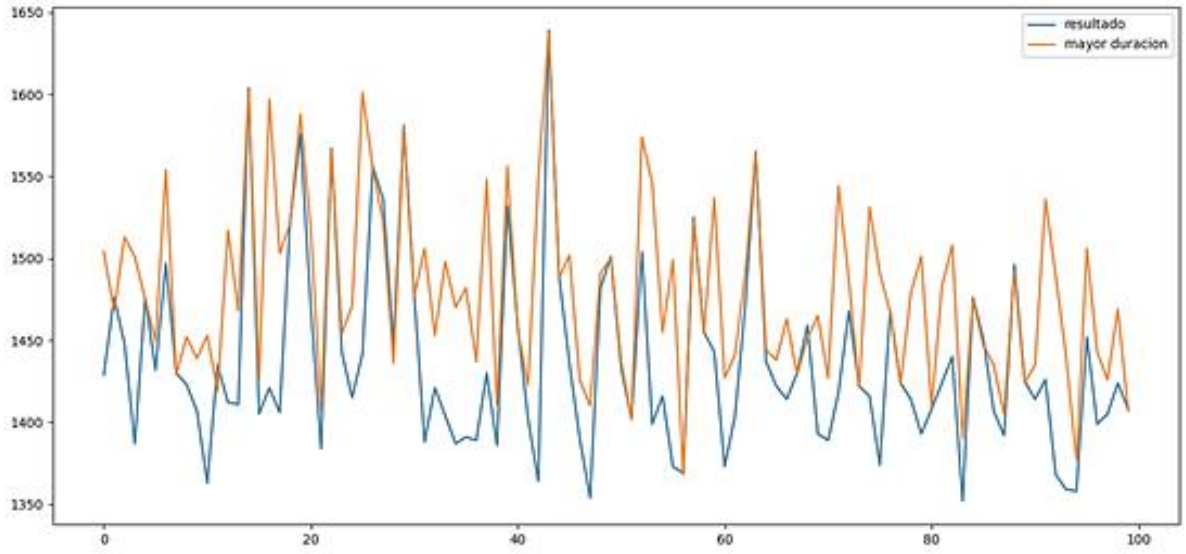
Luego de que a todas las maquinas que libres en el periodo actual se les hayan asignado una tarea se aumenta el periodo hasta que una maquina vuelva a estar libre, dado que como restricción solo pueden atender a una tarea a la vez.

El proceso se repite hasta que la lista de tareas se haya vaciado. Dicho proceso se puede apreciar de manera simplificada en el siguiente diagrama.

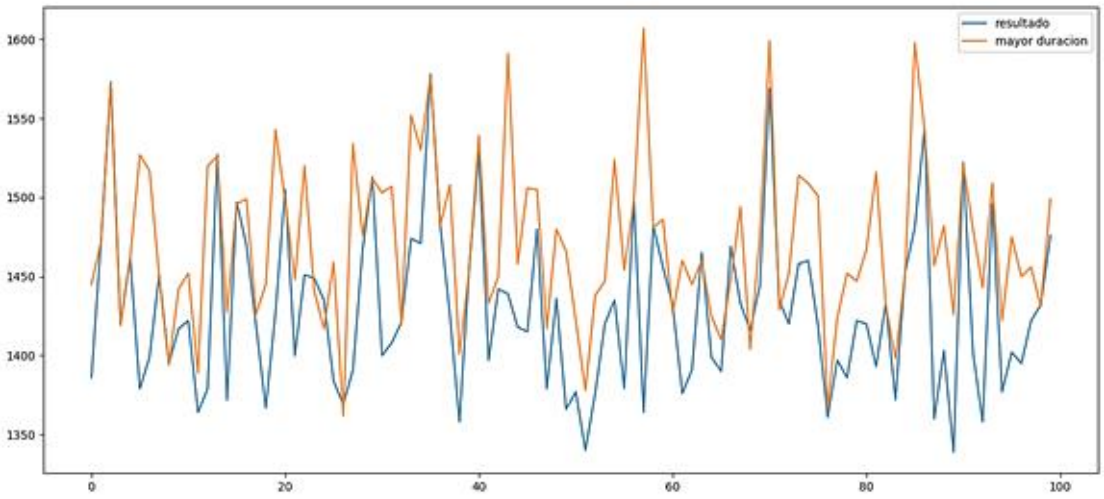
Muestras de ejecución

Dado que el resultado no es el óptimo al ser un algoritmo de búsqueda local. Tomamos como valor a comparar la sumatorio de las duraciones de todas las tareas que dieran el resultado de periodo más largo. El objetivo es que el algoritmo pudiera disminuir esa sumatorio de tiempo total de las tareas más largas.

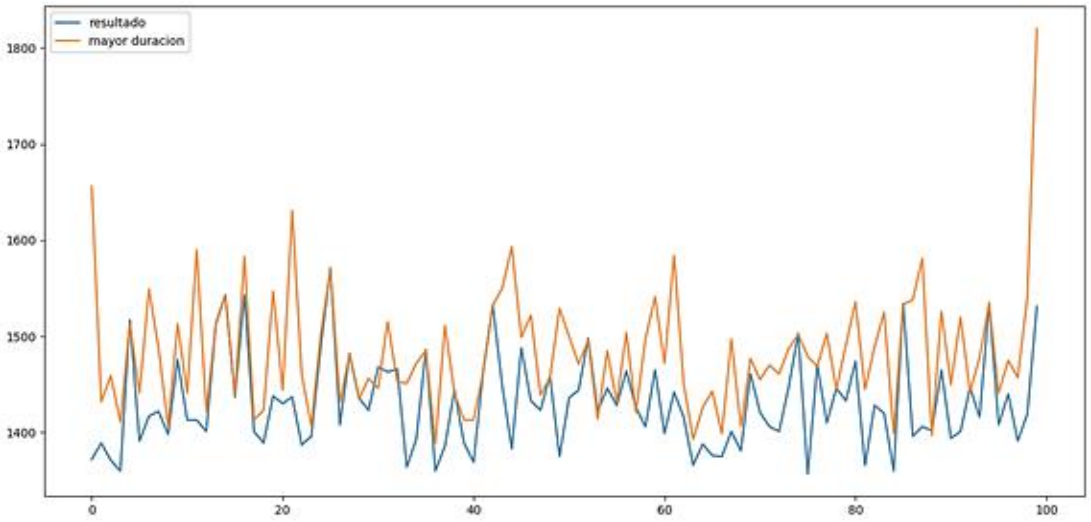
Para un problema de 1000 tareas, de duraciones y tipos aleatorios, se calculó el resultado de 100 ejecuciones del algoritmo y se graficaron los resultados.



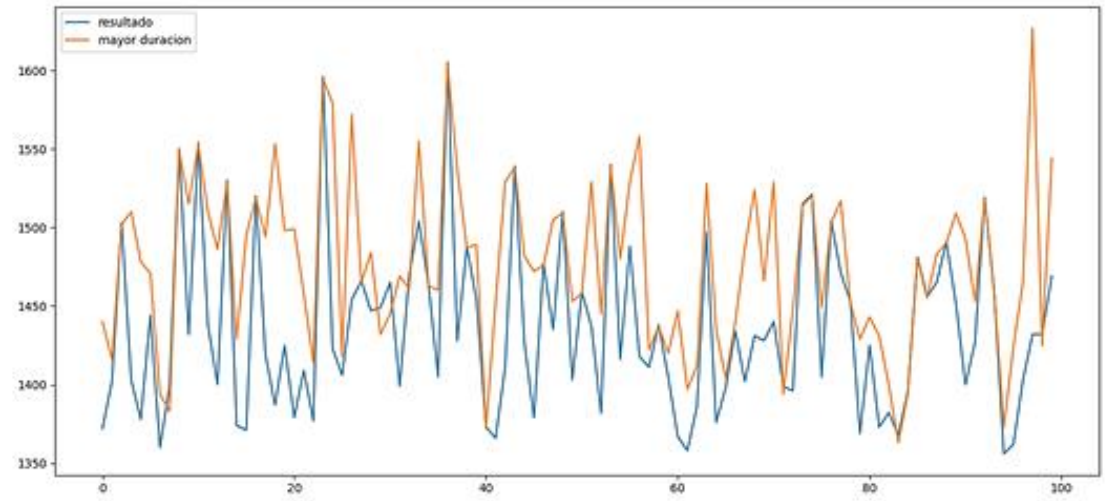

```
agustin@agustin-i500pro:~/NetBeansProjects/IA2/src$ /usr/bin/python3 /home/agustin/NetBeansProjects/IA2/src/ej7.py
Cuantas tareas aleatorias desea resolver?: 1000
promedio de resultados 1436.82
promedio de valores maximos 1478.89
```




```
agustin@agustin-i500pro:~/NetBeansProjects/IA2/src$ /usr/bin/python3 /home/agustin/NetBeansProjects/IA2/src/ej7.py
Cuantas tareas aleatorias desea resolver?: 1000
promedio de resultados 1428.89
promedio de valores maximos 1472.03
```




```
agustin@agustin-i500pro:~/NetBeansProjects/IA2/src$ /usr/bin/python3 /home/agustin/NetBeansProjects/IA2/src/ej7.py
Cuantas tareas aleatorias desea resolver?: 1000
promedio de resultados 1430.25
promedio de valores maximos 1483.66
```




```
PROBLEMS 3  OUTPUT  DEBUG CONSOLE  TERMINAL                    2: Python
agustin@agustin-i500pro:~/NetBeansProjects/IA2/src$ /usr/bin/python3 /home/agustin/NetBeansProjects/IA2/src/ej7.py
Cuantas tareas aleatorias desea resolver?: 1000
promedio de resultados 1435.61
promedio de valores maximos 1477.38
```

Como lo esperado encontramos casos en los cuales el algoritmo cumple muy bien su función dando como resultado tiempos muchos menores a los que demandaba la tarea más larga. Pero en otros encontramos tenemos tiempos similares a los de la tarea más larga y algunos pocos casos vemos un tiempo mayor a la misma.

Se calculó el promedio de los valores para determinar si en general estábamos teniendo una mejora o no. Como resultado los promedios de los tiempos de las tareas asignadas por el algoritmo siempre eran menores a las de la tarea de mayor duración. Por lo tanto determinamos que a la larga tenemos resultados favorables a un costo computacional muy bajo.

Dependiendo la aplicación de este algoritmo es si estamos dispuestos a correr el riesgo de usar resultados que no resultan los óptimos. Lo mejor sería pasar a un algoritmo de búsqueda global como el caso de A* con lo cual se asegura que el resultado además de completo sea optimo, pero esto demandaría una mayor complejidad de código y un mayor costo computacional dado que es un algoritmo en el cual se guardar un historial de búsqueda.