Trabajo Práctico N°1
Búsqueda y Optimización

2.a) Implemente el algoritmo A* y resuelva los siguientes problemas

- 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. Genere aleatoriamente los puntos de inicio y fin, y genere también aleatoriamente obstáculos que el robot debe esquivar, siempre en el espacio articular.

Para comenzar decidimos que todas las articulaciones podrían moverse 360° cada una, lo que podría ser fácilmente reemplazado utilizando una seed diferente para cada articulación según sus ángulos de movimiento definidos pero lo tomamos como simplificación del código

Luego, tal como dice la consigna, se generó aleatoriamente los puntos de inicio y final.

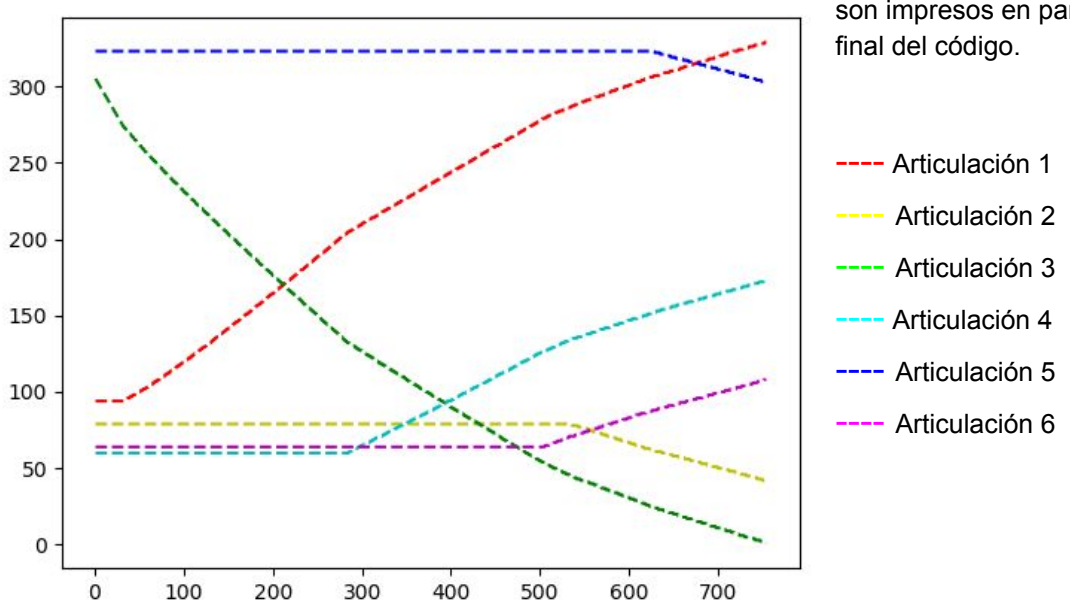
A continuación, en la función `Mov_Heurist`, creamos 99 obstáculos aleatoriamente (valor limitado por el tiempo que se tardará en crearlos y luego en su revisión). Entramos a un bucle while (línea 60), donde se verifica que aún no lleguemos a la posición final.

En el siguiente for (línea 65), recorremos los 6 grados de libertad del brazo y cambiamos su valor a un paso más arriba o uno más abajo, esto nos da 12 posibles nuevas posiciones, de las cuales elegiremos la más conveniente. Antes de esto verificamos con la función `flag` que ninguno de estos nuevos vecinos sea un obstáculo y también que ninguno de estos haya sido visitado anteriormente. Si cumple con estos dos requisitos `flag` devolverá un 3 y se guardará el nuevo vecino en el vector `hijos`, también se calculará su función heurística que depende de G, la distancia de la posición actual respecto a la inicial y de H, la distancia de la posición actual respecto a la final. Ambas son multiplicadas por un peso, los que variaremos para evaluar el tiempo que tarda el programa en encontrar la posición final.

Todos los costos (valores de función heurística) de cada vecino se acumula en el vector `resultados` del cual tomamos el menor y su correspondiente en `hijos` será la nueva posición actual.

El proceso se repite hasta que la posición actual sea igual a la posición final buscada.

Todos los datos relevante son impresos en pantalla final del código.

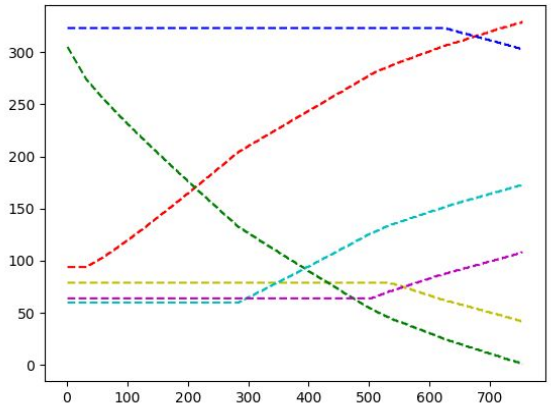


En esta figura vemos cada articulación en función de cada iteración que se realizó (en este caso 754) con 99 obstáculos y con peso de H = 1 y peso de G = 0,1. Tiempo : 30 seg aprox.
Posición inicial: [94 79 306 60 323 64] Posición final: [329 42 1 173 303 108]
Obstáculos Saltados: [ ]

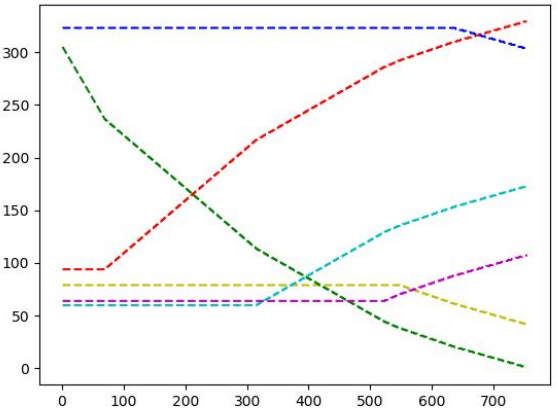
2) Con peso de H = 1 y peso de G = 0 Tiempo : 30 seg aprox.
1) Con peso de H = 1 y peso de G = 0,1. Tiempo : 30 seg aprox.
3) Con peso de H = 0.8 y peso de G = 0,4. Tiempo : 30 seg aprox
4) Con peso de H = 0.6 y peso de G = 0,25. Tiempo : 31 seg aprox.
5) Con peso de H = 0.5 y peso de G = 0,35. Tiempo : 31 seg aprox.
6) Con peso de H = 0.5 y peso de G = 0,45. Tiempo : 31 seg aprox.

Con igual posición inicial y final pero diferentes pesos, con 99 obstáculos y peso de H = 0.5 y peso de G = 0,5. El código llevaba 5 minutos corriendo y no lanzaba resultados, tampoco se modificaba correctamente, es decir, no se acercaba a la posición final.

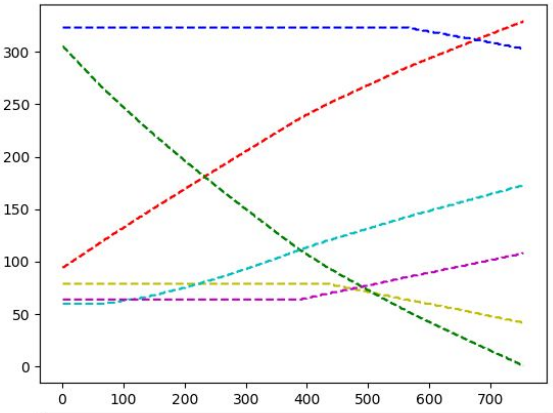
1)



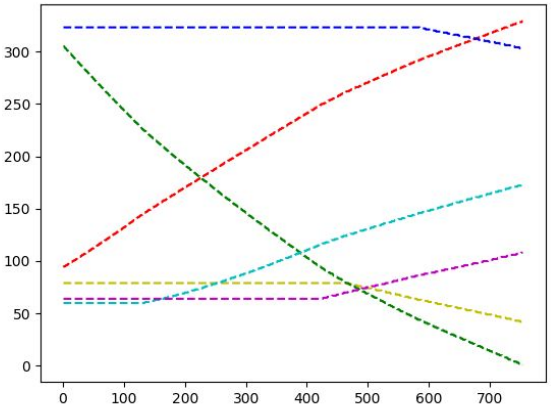
2)



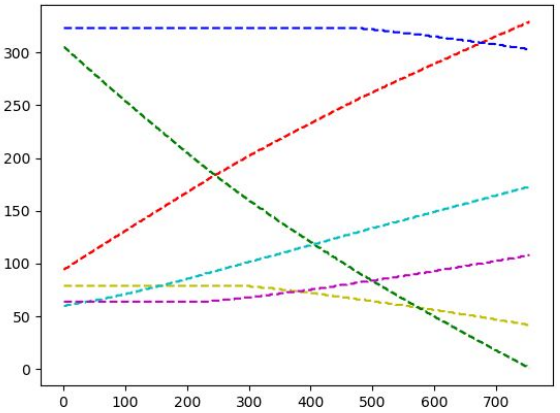
3)



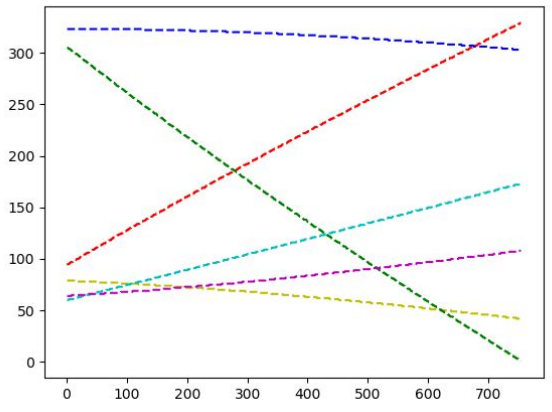
4)



5)



6)



2.b)

● 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

El primer paso fue crear una matriz con la forma del almacén mediante la función `createMatriz`, esta se consigue según el número de filas y columnas ingresadas. Esta matriz está llena de valores que corresponde al número del objeto colocado en ese estante. Aleatoriamente seleccionamos valor inicial y valor final. Estos son pasados a posición, por ejemplo el objeto de valor 1 está en la posición (0, 0).

Una vez hecho esto, llamamos a la función `Mov_Heurist`. Entramos a un bucle while (línea 62), donde se verifica que aún no lleguemos a la posición final.

En el siguiente for (línea 66), recorremos con i=0 la primera coordenada, es decir, nos moveremos hacia arriba o hacia abajo. Obtenemos así, dos vecinos que se guardan en la lista de vectores `hijos`, se calculará su función heurística (costo, `F`) que depende de G, que devuelve una distancia de 1 si nos movemos hacia una posición que no sea una esquina y raiz de 2 si lo es. También de H, la distancia de la posición actual respecto a la final. Este costo se acumula en la lista `resultados`.

Cuando en el for la i pasa a ser 1, se refiere a la segunda coordenada que nos indica que nos moveremos hacia la derecha o a la izquierda (mismo procedimiento con `hijos`, `F` y `resultados`).

Allí entramos en un if (línea 81) donde el primer bloque nos da un vecino Arriba - Izquierda, el segundo Abajo - Izquierda, el tercero Arriba - Derecha y el cuarto Abajo - Derecha (mismo procedimiento con `hijos`, `F` y `resultados`).

Esto nos da 8 vecinos, de los cuales elegimos el que tenga la menor función heurística que será nuestro nuevo vector posición.

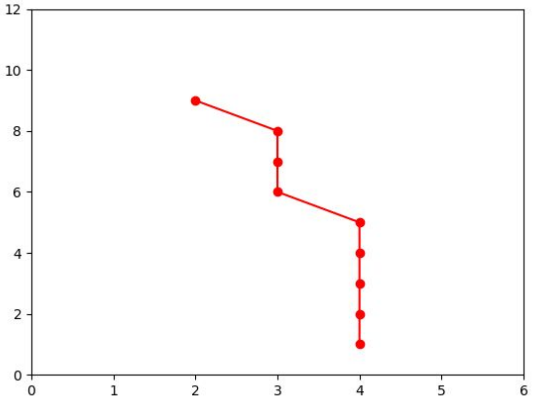
El proceso se repite hasta que la posición actual sea igual a la posición final buscada.

En el for de la línea 142, pasamos la lista de movimientos realizados de posición a valor y también calculamos la distancia recorrida. En la línea 155 calculamos la distancia euclidiana entre el punto inicial y el final.

Todos los datos relevante son impresos en pantalla al final del código.

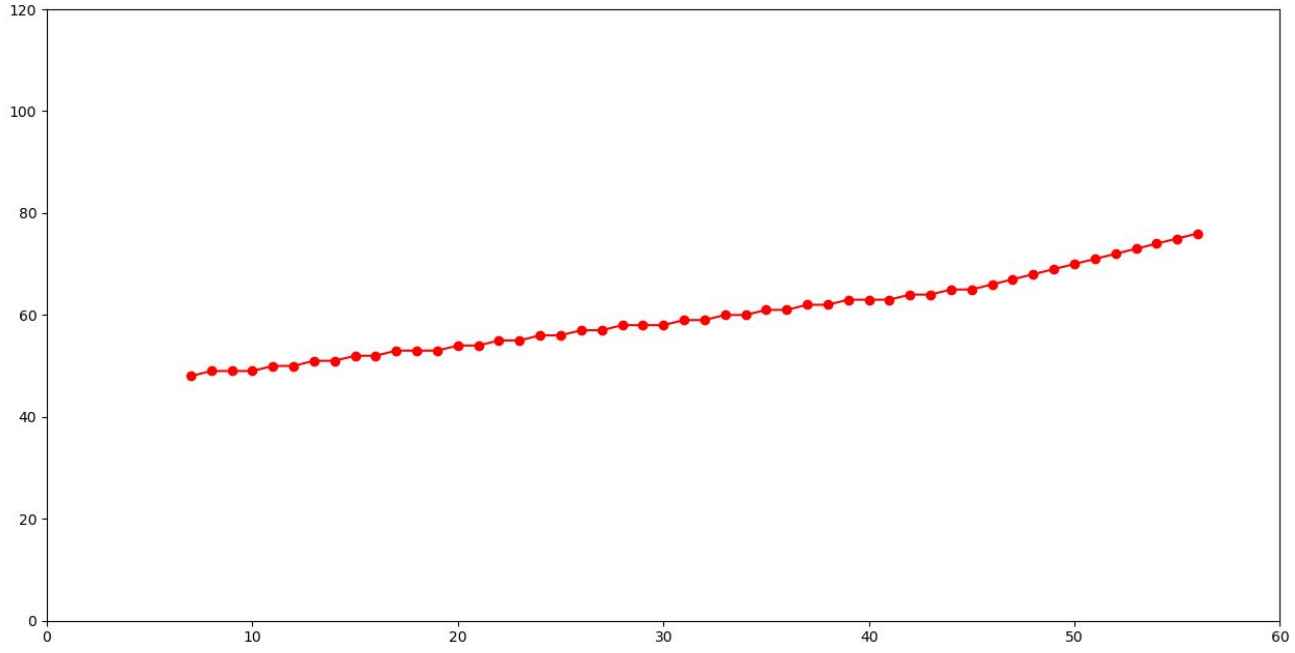
Valor inicial: 51
Posición inicial: [1, 4]
Movimientos realizados: [[1 4] [2 4] [3 4] [4 4] [5 4] [6 3] [7 3] [8 3] [9 2]]
Valores de los movimientos realizados: [51 53 55 57 59 38 40 42 43]
Distancia entre punto inicial y final (euclidiana): 8.246211251235321
Distancia que se recorrió: 8.82842712474619
Valor final esperado: 43
Posición final esperada: [9, 2]
Valor final encontrado: 43
Posición final encontrada: [9, 2]
Filas:12
Columnas: 6

Matriz inicial:
[[1. 2. 25. 26. 49. 50.]
[3. 4. 27. 28. 51. 52.]
[5. 6. 29. 30. 53. 54.]
[7. 8. 31. 32. 55. 56.]
[9. 10. 33. 34. 57. 58.]
[11. 12. 35. 36. 59. 60.]
[13. 14. 37. 38. 61. 62.]
[15. 16. 39. 40. 63. 64.]
[17. 18. 41. 42. 65. 66.]
[19. 20. 43. 44. 67. 68.]
[21. 22. 45. 46. 69. 70.]
[23. 24. 47. 48. 71. 72.]]



Variando número de filas y columnas:

Valor inicial: 6873
Posición inicial: [76, 56]
Movimientos realizados:[[76 56] [75 55] [74 54] [73 53] [72 52] [71 51] [70 50] [69 49] [68 48] [67 47] [66 46] [65 45] [65 44] [64 43] [64 42] [63 41] [63 40] [63 39] [62 38] [62 37] [61 36] [61 35] [60 34] [60 33] [59 32] [59 31] [58 30] [58 29] [58 28] [57 27] [57 26] [56 25] [56 24] [55 23] [55 22] [54 21] [54 20] [53 19] [53 18] [53 17] [52 16] [52 15] [51 14] [51 13] [50 12] [50 11] [49 10] [49 9] [49 8] [48 7]]
Valores de los movimientos realizados: [6873 6632 6629 6388 6385 6144 6141 5900 5897 5656 5653 5412 5411 5170 5169 4928 4927 4688 4685 4446 4443 4204 4201 3962 3959 3720 3717 3478 3477 3236 3235 2994 2993 2752 2751 2510 2509 2268 2267 2028 2025 1786 1783 1544 1541 1302 1299 1060 1059 818]
Distancia entre punto inicial y final (euclidiana): 56.43580423808985
Distancia que se recorrió: 60.59797974644664
Valor final esperado: 818
Posición final esperada: [48, 7]
Valor final encontrado: 818
Posición final encontrada: [48, 7]
Filas: 120
Columnas: 60



La diferencia de tiempo entre ambos es imperceptible.

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?

Comenzamos declarando las variables necesarias para realizar el Temple Simulado.

Hacemos que el valor que está en la posición (0, 0) se convierta en nuestra bahía de carga (en este ejercicio siempre será el número 1).

Creamos aleatoriamente la lista con 10 pedidos (pueden cambiar) con la función `crear_lista_pedidos`, la cual también nos coloca la bahía de carga al principio y al final del vector `pedidos`.

Entramos en el bucle while necesario para realizar el Temple, este parará cuando la temperatura actual sea igual a la final establecida.

Luego, reordenamos la lista de pedidos, es decir, elegimos un elemento aleatorio y lo cambiamos por su vecino con la función `random_vecinos`. Sacamos el costo de esta nueva lista que está dado por la distancia total recorrida, que es calculada con el ejercicio 2 (Astar). La función `dist_total` realiza una búsqueda A estrella para cada par en la lista, es decir, comienza desde la bahía de carga hasta el siguiente valor en pedidos, de ese, al tercero y así hasta volver a la bahía. Todas las distancias recorridas se suman en `dist`, que nos da nuestro nuevo costo.

Sacamos la `diferencia` entre el costo nuevo y el anterior (si es negativo, el nuevo es mejor, o sea, menor).

Ahora entramos en la condición, si este valor `diferencia` es negativo o si `e^(-diferencia/temperatura)` es mayor a cierto número aleatorio entre 0 y 1, entonces el costo nuevo es aceptado y pasa a ser nuestro costo anterior (esto nos da la oportunidad de aceptar valores que son peores pero que tal vez conduzcan a una solución mejor más adelante, hay que tener en cuenta que esto depende mucho de la diferencia y de la temperatura). También agregamos esa lista de pedidos a la lista `all_caminos` y el costo a la lista `all_costos`.

Luego de esto bajamos la temperatura exponencialmente (a nuestra elección).

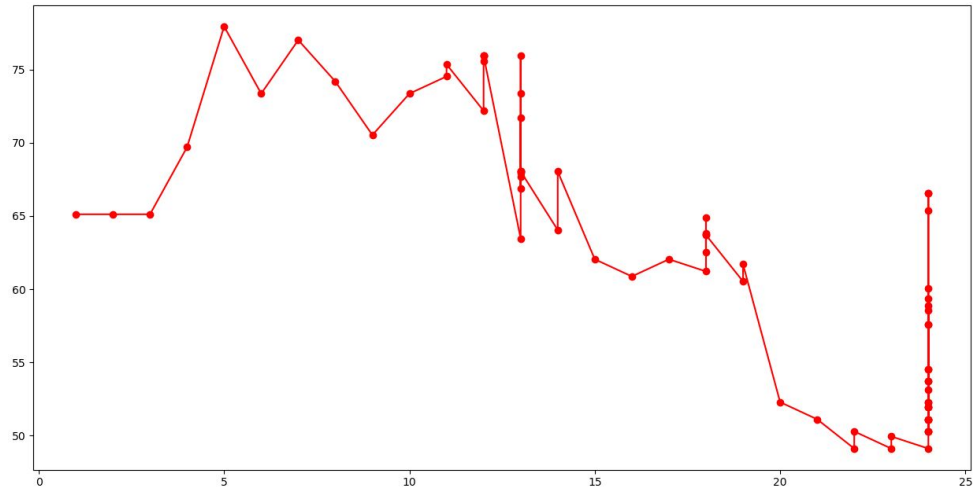
Una vez que la temperatura llega a su valor final, tendremos un vector con todos las diferentes listas (vecinos) visitados y sus costos, entonces elegimos el menor y lo tomamos como solución final.

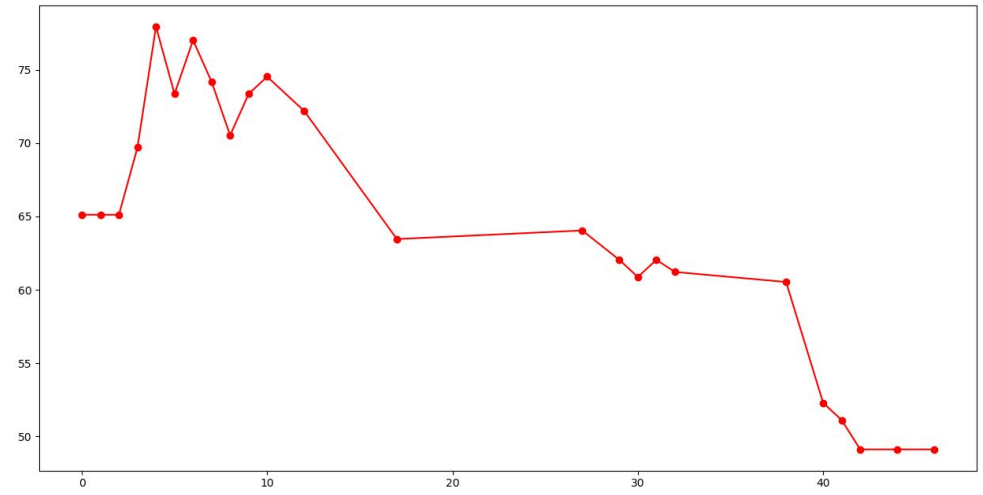
Todos los datos relevante son impresos en pantalla al final del código.

```
Caminos observados:  73
Costo de todos los caminos aceptados:  [65.11269837220809, 65.11269837220809,
65.11269837220809, 69.698484809835, 77.94112549695427, 73.35533905932736,
77.01219330881976, 74.18376618407356, 70.52691193458118, 73.35533905932736,
74.52691193458118, 72.18376618407356, 63.45584412271571, 64.04163056034261,
62.04163056034262, 60.8700576850888, 62.04163056034262, 61.21320343559643,
60.52691193458119, 52.2842712474619, 51.112698372208094, 49.112698372208094,
49.112698372208094, 49.112698372208094]
El mejor camino es el número:  22
Mejor costo:  49.112698372208094
Mejor camino:
 [[ 1. 65. 59. 14. 51. 54. 50. 10. 47. 21. 59.  1.]]
Temperatura inicial:  20
Temperatura final:  0.01
Factor de enfriamiento:  0.9
Primera Lista: [ 1. 65. 14. 54. 59. 50. 21. 10. 59. 47. 51.  1.]
```

En la primera imagen vemos los costos de todas las listas vistas, hasta esas que no fueron aceptadas, esto se puede ver en donde hay varios puntos en una línea recta, es decir, no fueron aceptados.

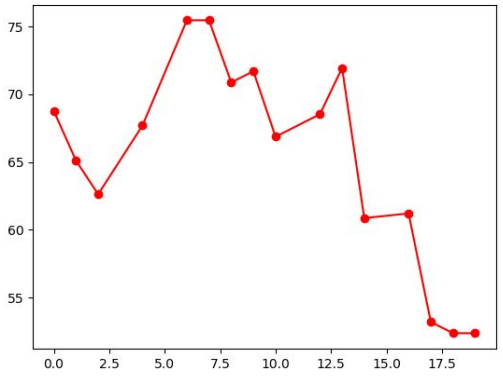
En la segunda imagen se puede ver cómo funciona este algoritmo de búsqueda, se superaron varios mínimos locales y mesetas, vemos que de otro modo nunca hubiéramos alcanzado estos mínimos inferiores, que si bien no sabemos si son globales, pero al menos obtuvimos una solución mejor.



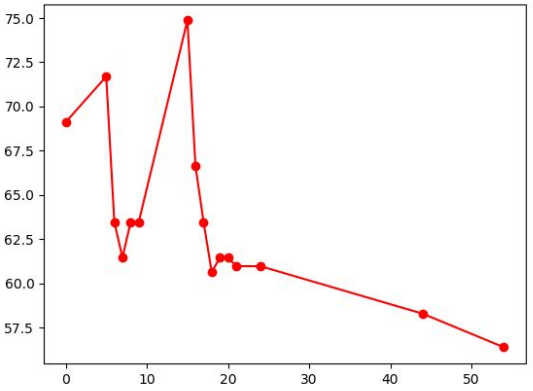


CAMBIAMOS TEMPERATURA FINAL (misma lista inicial)

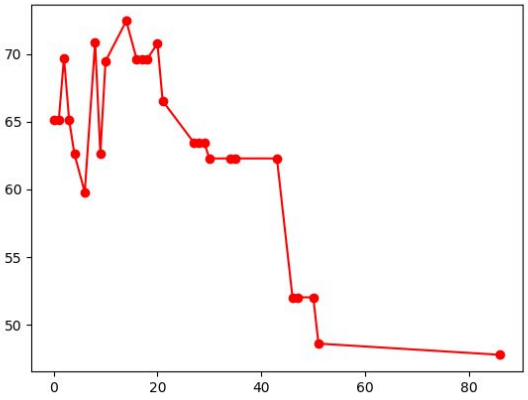
Caminos observados: 51
Mejor costo: 52.38477631085023
Mejor camino: [[1 54 59 14 47 65 51 50 59 10 21 1]]
Temperatura inicial: 20
Temperatura final: 0.1
Factor de enfriamiento: 0.9



Caminos observados: 94
Mejor costo: 56.384776310850235
Mejor camino: [[1 10 54 21 65 14 59 47 50 59 51 1]]
Temperatura inicial: 20
Temperatura final: 0.001
Factor de enfriamiento: 0.9



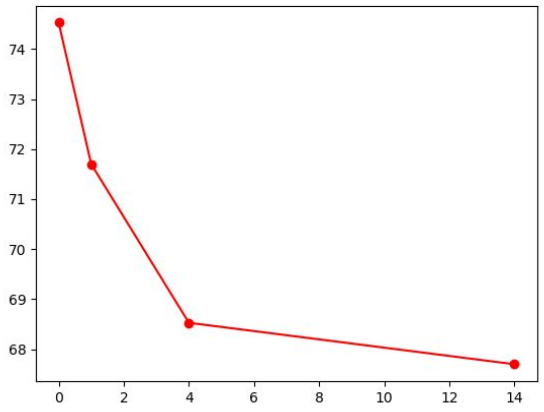
Caminos observados: 116
Mejor costo: 47.79898987322333
Mejor camino: [[1 54 14 65 59 59 21 47 51 10 50 1]]
Temperatura inicial: 20
Temperatura final: 0.0001
Factor de enfriamiento: 0.9



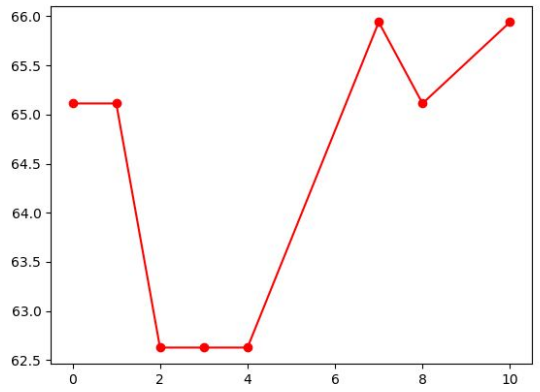
Vemos como aumenta la cantidad de caminos observados al disminuir la temperatura final.
Mientras más listas se observen, más probabilidades de encontrar un mínimo menor hay.

CAMBIAMOS FACTOR DE ENFRIAMIENTO (misma lista inicial)

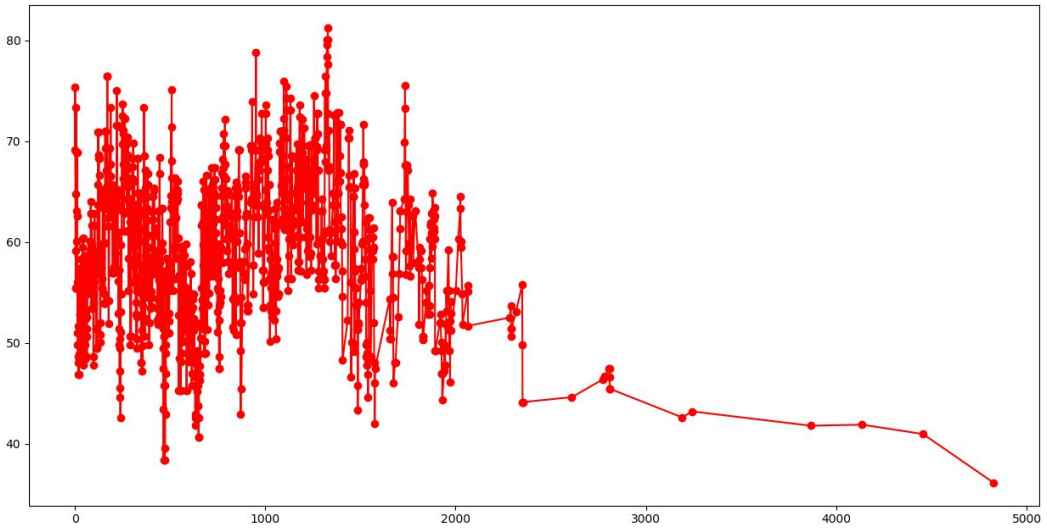
Caminos observados: 15
Mejor costo: 67.698484809835
Mejor camino: [[1 14 65 50 54 21 51 10 59 59 47 1]]
Temperatura inicial: 20
Temperatura final: 0.001
Factor de enfriamiento: 0.5



Caminos observados: 35
Mejor costo: 62.62741699796952
Mejor camino: [[1 65 59 21 54 59 50 14 51 47 10 1]]
Temperatura inicial: 20
Temperatura final: 0.001
Factor de enfriamiento: 0.75



Caminos observados: 9899
Mejor costo: 36.14213562373095
Mejor camino: [[1 54 51 47 21 10 59 65 14 50 59 1]]
Temperatura inicial: 20
Temperatura final: 0.001
Factor de enfriamiento: 0.999



Vemos como aumenta la cantidad de caminos observados al aumentar el factor de enfriamiento. Mientras más listas se observen, más probabilidades de encontrar un mínimo menor hay.