Trabajo Práctico Nº 1 de Inteligencia Artificial 2
Búsqueda y Optimización

Año 2022
- Cabrero García, Gabriel
- Mellimaci, Marcelo E
- Tous Maggini, Martín

Repositorio: https://github.com/MartinTous/Practica-IA2.git

Algoritmo de búsqueda A*

La finalidad de este algoritmo es poder encontrar el camino óptimo entre dos puntos dentro de un almacén, para ello, se utilizó el algoritmo A*. Para llevarlo a cabo el programa se encuentra seccionado en dos módulos, main y A*.

En el main se ejecuta el algoritmo y el mismo está compuesto de subfunciones tales como ubicación y almacén, explicadas a continuación.

almacen(matriz, dim)
```
Con esta funcion generamos la matriz que representa al almacén,
para ello debemos pasarle la dimension de la matriz y genera un almacén cuadrado [dim x dim]
```

ubicacion(matriz, pos, dim)
```
Esta funcion se encarga de recibir el producto en cuestion,
buscarlo en la matriz almacén y devolver las coordenadas del mismo
```

Una vez generado el almacén y buscar las coordenadas de inicio y destino, se llama a la función A estrella, la cual está formada por las siguientes subfunciones.

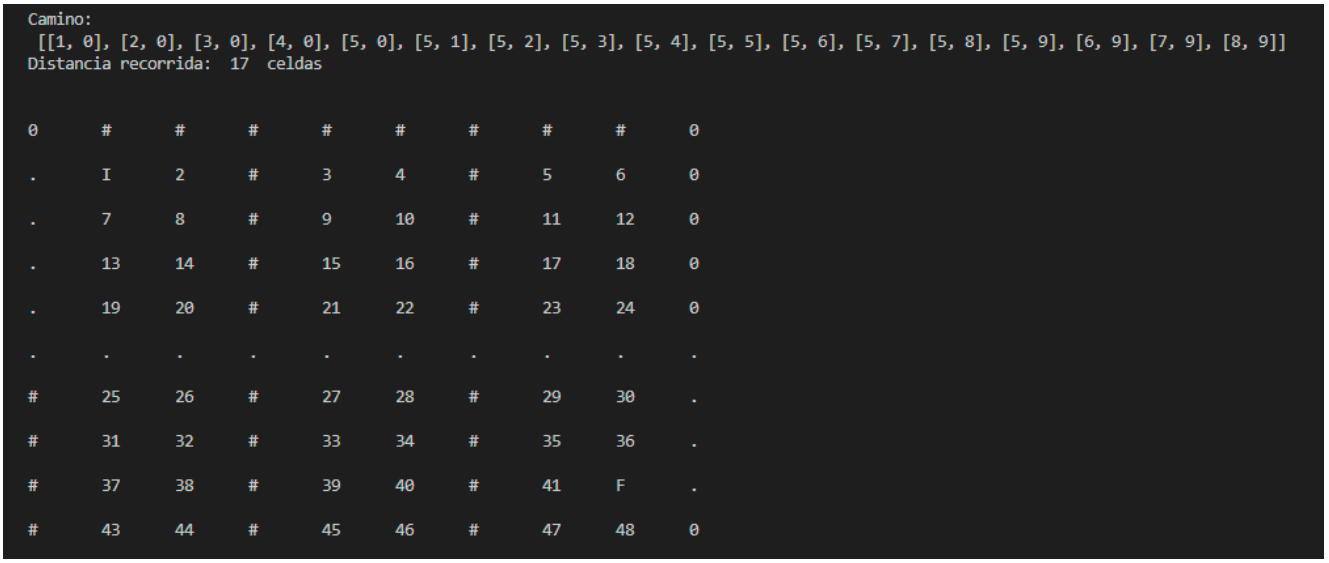
Astar(matriz, inicio, destino)
```
Algoritmo que encuentra el camino óptimo entre el inicio y destino, para eso pasamos como argumento
la matriz que representa al almacén, y las coordenadas del inicio y destino dentro del almacén
```

fn(pSig, destino, costo)
```
Funcion que calcula el f(n)=c(n)+h(n), es decir el costo más la heurística
Para ello, ingresamos la posición siguiente a visitar, el destino donde se debe
llegar y el costo para llegar desde el inicio hasta la siguiente posicion en cuestion
```

Más allá de aspectos técnicos específicos del programa, se ha podido concluir que es altamente eficaz en el entorno en el que se ejecuta (matriz almacén) obteniendo así, en el 100% de los casos probados la ruta más corta posible.

En cuanto a la eficiencia del mismo, aún se cree que podría mejorarse, ya que cuando se lo lleva a gran escala, es decir, gran cantidad de órdenes para ejecutar y muchos caminos que encontrar, suele demorarse varios segundos.

A continuación un ejemplo de los resultados obtenidos con este algoritmo.



En este caso, se le indicó la posición de inicio (Pinicio=1) y la posición de destino (Pdestino=42).
La línea de puntos (.) representa el camino definitivo para llegar de la posición de inicio hasta el destino, mientras que los caminos con numerales (#), representa los caminos que el algoritmo recorre pero que no fueron los óptimos, por ende los rechaza.
Además, muestra las coordenadas del camino óptimo y el costo del mismo, es decir, la cantidad de celdas para llegar al destino.

Algoritmo de Recocido Simulado

Obtención Del Orden Óptimo De Picking Mediante Recocido Simulado

Dada una orden de pedido, que incluye una lista de productos del almacén anterior que deben ser despachados en su totalidad, esta implementación determina el orden óptimo para la operación de picking para una serie de productos en un almacén mediante el algoritmo de Temple Simulado ó Recocido Simulado.
Para resolver un problema de este tipo también podría haber sido posible usar algoritmos genéticos (AG), tomando una función de fitness inversamente proporcional a la distancia total recorrida para una orden de pedido determinada
Los parámetros de entrada que del programa en cuestión son:

- Lista de Picking: Una lista con números que corresponden a cada uno de los productos del almacén, con un ordenamiento inicial cualquiera. Dicha lista es tomada de una de las órdenes de pedido del archivo de texto orders.txt proporcionado por la cátedra. Mientras mayor sea la cantidad de ítems en una orden, mayor será el tiempo de ejecución del algoritmo
- Plano del Almacén: Matriz bidimensional del mapa del almacén con un número único para cada posición del almacén donde se halla cada uno de los distintos ítems de la

lista, y en el cual los elementos que valen cero indican un pasillo en el cual es posible circular

- Temperatura Inicial (To): La variable temperatura disminuye desde un valor inicial positivo alto hasta un valor menor a los anteriores. La temperatura inicial debe ser bastante elevada para evitar que el resultado del algoritmo se quede atrapado en un óptimo local (mínimo local para el caso analizado), pero se debe tener en cuenta que una mayor temperatura inicial implica un mayor número de iteraciones y un tiempo de ejecución más largo. No se observó una mejora notoria de las soluciones obtenidas con temperaturas iniciales mayores a $To = 1500°$
- Velocidad de Enfriamiento (α): Determina cuánto se reducirá el parámetro de temperatura en cada iteración, ya que en el algoritmo implementado se adoptó una curva de enfriamiento exponencial que irá tendiendo a cero asintóticamente. Esta velocidad de enfriamiento α(alfa) es una constante entre 0 y 1 que multiplica al valor de temperatura de la iteración anterior para obtener la temperatura de una iteración determinada
- Temperatura Final (Tf): Valor de temperatura luego del cual concluirá el algoritmo. Este parámetro deberá ser mayor a cero, ya que debido al patrón elegido para el descenso de la temperatura nunca se alcanzará una temperatura de 0° para un número finito de iteraciones. Por ejemplo, podría llegar a ser: 0.01°

Las variables de salida que devuelve el programa son:

- Lista Ordenada: Listado ordenado de forma tal de reducir la distancia recorrida al buscar los productos del listado
- Distancia Recorrida: desplazamiento minimizado por el algoritmo

En cada iteración del algoritmo, se tendrá un estado que es una lista con la secuencia de productos a seleccionar

Para la lista de picking obtenida en cada una de las iteraciones, se calcula la longitud total del camino óptimo dentro del almacén. Se obtiene menor distancia posible para ir de cada coordenada a la siguiente a través del algoritmo A estrella desarrollado en el ejercicio uno, y luego la suma de cada uno de esos respectivos desplazamientos dará la trayectoria óptima, lo cual nos dará el valor de energía (E) para dicho estado

Para generar estados vecinos se seleccionan al azar dos elementos del listado de productos y se intercambian de lugar en la secuencia (el par de productos escogidos aleatoriamente no necesariamente deben ser elementos contiguos en la lista)

Si la energía de dicho estado vecino es menor (lo cual implicaría que esa posible solución es mejor que la anterior), esta nueva secuencia de productos se acepta como estado sucesor. Por lo contrario, si esta nueva solución es peor (ΔE > 0) se utilizará la probabilidad de Boltzmann para lograr que mientras menos mala sea la solución obtenida más probable sea que la aceptemos. Dicha posibilidad se obtiene a través de la siguiente ecuación:

$$probabilidad = e^{-\Delta E / T}$$

Si bien el resultado obtenido mediante el algoritmo de Simulated Annealing puede no llegar a ser con exactitud el “óptimo global” (es decir, no será el mejor), será una buena aproximación del mismo

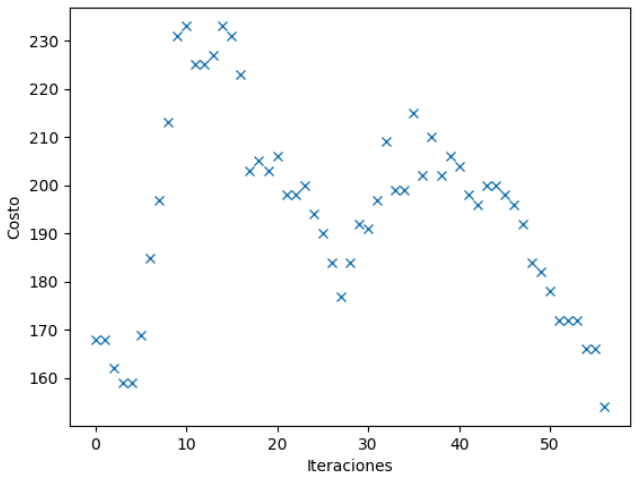
Se crearon documentos HTML para cada uno de los módulos de código del programa, los cuales están disponibles en:
https://drive.google.com/file/d/1JhbJjFx_S3cQBArzRFAjI1FgJWeOv_Zq/view

También se tomaron como ejemplos cuatro casos para los cuales se usó el programa descripto:

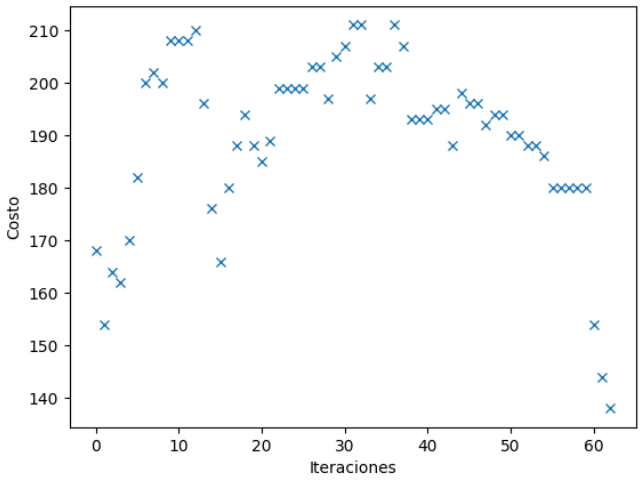
https://drive.google.com/drive/folders/1XFn7JS4Zsr12LtteJN59VrVRJurxbFzV

A continuación se han graficado algunos de los resultados obtenidos con este programa:

Orden 1

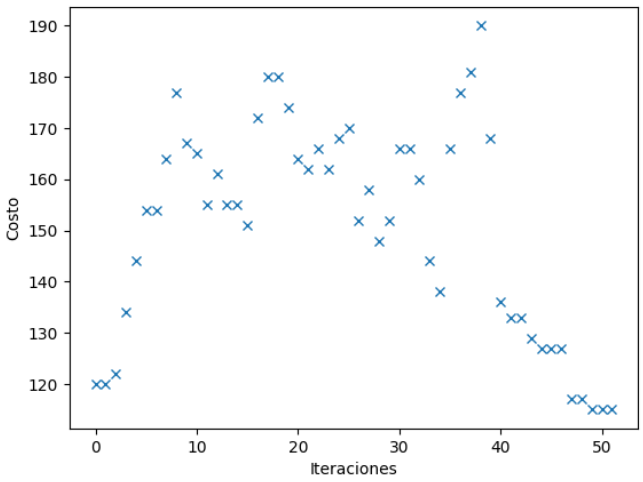


Costo final: 154

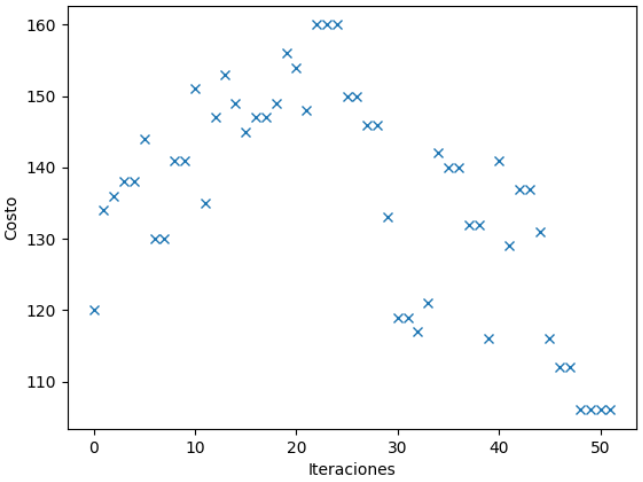


Costo final: 138

Orden 30



costo final: 115



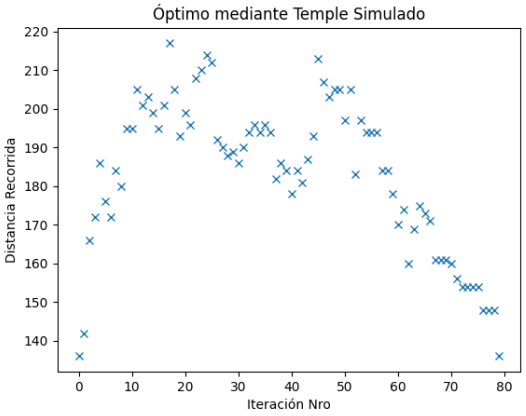
Costo final: 106

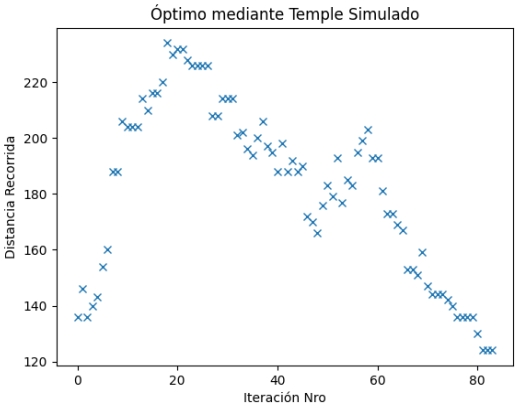
Se puede observar claramente cómo el algoritmo al comienzo selecciona valores peores pero a medida que avanza, la probabilidad de que tome un valor peor disminuye, por ende los valores seleccionados comienzan a mejorar y a disminuir sus respectivos costos.

Además, se puede mejorar el resultado usando otra vez el recocido simulado, de forma tal que la salida de un recocido simulado sea la entrada del segundo, e.g.

```
$ python3 main.py
     Ejercicio 2 del TP 1 de IA 2


[[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   1   2   0   3   4   0   5   6   0   7   8   0   9  10   0]
 [  0  11  12   0  13  14   0  15  16   0  17  18   0  19  20   0]
 [  0  21  22   0  23  24   0  25  26   0  27  28   0  29  30   0]
 [  0  31  32   0  33  34   0  35  36   0  37  38   0  39  40   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0  41  42   0  43  44   0  45  46   0  47  48   0  49  50   0]
 [  0  51  52   0  53  54   0  55  56   0  57  58   0  59  60   0]
 [  0  61  62   0  63  64   0  65  66   0  67  68   0  69  70   0]
 [  0  71  72   0  73  74   0  75  76   0  77  78   0  79  80   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0  81  82   0  83  84   0  85  86   0  87  88   0  89  90   0]
 [  0  91  92   0  93  94   0  95  96   0  97  98   0  99 100   0]
 [  0 101 102   0 103 104   0 105 106   0 107 108   0 109 110   0]
 [  0 111 112   0 113 114   0 115 116   0 117 118   0 119 120   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]]
Indique el nro [1 ; 100] de la orden: 100
El archivo ingresado para tomar la lista de picking de la orden es:
ordenes/order_100.txt
Lista de picking sin ordenar:
[24, 33, 39, 40, 41, 42, 44, 51, 62, 66, 67, 69, 71, 72, 85, 87, 92, 94, 95,
96, 97, 98]
Distancia recorrida con la lista sin ordenar:
136
Lista ordenada para reducir la distancia recorrida:
[87, 96, 97, 67, 95, 39, 44, 69, 98, 40, 42, 24, 33, 62, 72, 71, 51, 41, 92,
66, 94, 85]
Distancia minimizada con la lista ordenada:
136
Quiere optimizar otra vez para mejorar la calidad del resultado ???
[S/n] S
Lista ordenada nuevamente para reducir la distancia recorrida:
[87, 96, 97, 67, 95, 39, 44, 69, 98, 40, 42, 24, 33, 62, 72, 71, 51, 41, 92,
66, 94, 85]
Distancia minimizada con la lista ordenada:
136
Programa terminado
```





Algoritmos genéticos

Descripción

En este algoritmo se intenta encontrar una disposición óptima de los productos en el almacén. Para esto se usa como base un historial de órdenes.

Programa

class **individuo**(builtins.object)

```
individuo(lista, ordenes)

Clase Individuo:
Clase a partir del cual creamos objetos Individuos que contienen como atributos una lista
de disposicion y un fitness asiciado.
```

setfitness(self, ordenes)

```
Método setfitness:
Se encarga de calcular el atributo fitness a partir de llamar al algoritmo recocido simulado
y acumular los costos de cada orden.
```

genetico(disp, ordenes)

```
Funcion genetico:
Recibe una poblacion inicial de disposicones y un conjunto de ordenes.
Realiza las operaciones propias, y devuelve la disposión mas eficiente que encontró
```

seleccion(poblacion)

```
Funcion seleccion:
Recibe una poblacion y selecciona la mitad con mejor fitness para que
conforme la siguiente generación.
```

crossover(lista)

```
Funcion crossover:
Recibe una poblacion seleccionada incompleta y la rellena realizando
operaciones de crossover a estos.
```

```
mutacion(poblacion)
      Funcion mutacion:
      A los nuevos individuos creados a partir de crossover, le aplica una mutación de 3 puntos
```

Observaciones

Durante la implementación del algoritmo en lenguaje python se destaca el tiempo empleado para diseñar una aplicación del crossover. El mismo que es del tipo cruce de orden, tiene complejidad a la hora de revisar que el gen a insertar no se encuentre ya en el individuo.

En la implementación de una función para mutar individuos no se encontró dificultad alguna por la sencillez del caso.

Como criterio de parada se utilizó la cantidad de iteraciones, por temas de tiempo. A modo de mejora se podría incluir un criterio adicional que revise la mejora de una generación a otra y su variabilidad. Así en caso de encontrar una buena solución, el algoritmo corta antes de cumplir las iteraciones, ahorrando tiempo y costo computacional.

El mayor problema estuvo en el tiempo que demanda calcular los costos con los otros programas En un primer momento el algoritmo genético llamaba a recocido simulado y este a su vez al A* para poder calcular el costo. De esta forma hacer correr el recorrido simulado tardaba 40 segundos aproximadamente. Teniendo en cuenta que llamamos a esta función una vez por cada orden con cada individuo, el tiempo invertido sería demasiado. Para esto se decidió dejar de usar A* en cada llamado, y sustituirlo con un archivo que contenga las distancias ya calculadas. Esto redujo el tiempo considerablemente, a tal punto de tardar 8 segundos en el análisis de cada población. A pesar de la notable reducción, sigue siendo ineficiente el tiempo por lo que a continuación, se pasó a dejar abierto el archivo que contiene las distancias.

Resultados

A continuación se gráfica el mejor individuo de cada generación (menor fitness). Con esto se observa como va evolucionando cada generación, obteniendo individuos con mejores fitness. Como aspecto a mejorar, se puede graficar la generación completa, para así poder observar la variabilidad de las generaciones.