Universidad Nacional de Cuyo - Facultad de Ingeniería

Búsqueda y Optimización

Documentación del Trabajo Práctico N°1

Cátedra: Inteligencia Artificial II

Grupo 2:
Gaviño, Matias - 13019
Mercado, Tomas - 12640
Sotar Maidana, D. Maximiliano - 12231
Vitelli, Gabriel N.- 13001

Índice

I. Introducción

a. Objetivos Generales

Desarrollar una solución de optimización para mejorar el proceso de picking en un almacén dado, utilizando diferentes algoritmos.

b. Objetivos específicos:

- Calcular el camino más corto y la distancia entre dos posiciones del almacén, utilizando un algoritmo A* adecuado.
- Determinar el orden óptimo de picking para una lista de productos de un pedido dado, utilizando el algoritmo de Temple Simulado, así como también explorar otros algoritmos que puedan ser utilizados para esta tarea.
- Implementar un algoritmo genético para resolver el problema de optimizar la ubicación de los productos en el almacén, con el fin de mejorar el proceso de picking y reducir el costo asociado a la distancia recorrida.

c. Alcances

- El trabajo práctico aborda el problema de optimización del proceso de picking en un almacén específico, asumiendo un layout fijo.
- La solución propuesta se enfoca en tres aspectos clave del proceso de picking: cálculo de la ruta más corta entre dos posiciones, determinación del orden óptimo de picking para una lista de productos y optimización de la ubicación de los productos en el almacén.
- La solución propuesta considera que el costo asociado a la distancia recorrida es proporcional al costo del picking.
- Se explorarán diferentes algoritmos para resolver cada uno de los problemas específicos planteados.

d. Descripción de los programas desarrollados

- Para la primera consigna, se ha desarrollado un programa que utiliza un algoritmo A* para calcular el camino más corto y la distancia entre dos posiciones del almacén, dadas sus coordenadas. Este programa recibe como entrada el layout del almacén, los costos asociados a las coordenadas dentro del mismo y las coordenadas de las dos posiciones, y devuelve como salida el valor de la distancia del camino más corto, así como también un archivo de texto CSV con las distancias recorridas entre las distintas coordenadas.
- Para la segunda consigna, se ha desarrollado un programa que utiliza el algoritmo de Temple Simulado para determinar el orden óptimo de picking para una lista de productos de un pedido dado. Este programa recibe como entrada el layout del almacén y la lista de productos a despachar, y devuelve como salida el orden óptimo de picking para minimizar el costo asociado a la distancia recorrida.

- Para la tercera consigna, se ha desarrollado un programa que utiliza un algoritmo genético para optimizar la ubicación de los productos en el almacén y mejorar el proceso de picking. Este programa recibe como entrada el layout del almacén y la lista de productos a despachar, y devuelve como salida la ubicación óptima de los productos en el almacén para minimizar el costo asociado a la distancia recorrida durante el proceso de picking. El programa utiliza diferentes técnicas de algoritmos genéticos, como selección, cruzamiento y mutación, para encontrar la solución óptima.

II. Desarrollo

a. Descripción general de los algoritmos implementados

A continuación se presenta una descripción simplificada de la relevancia de cada uno de los algoritmos implementados:

1. A_estrella.py:

   Este script implementa el algoritmo de búsqueda A* para encontrar la ruta más corta entre dos puntos en un mapa.

2. AlgoritmoGenético.py:

   Este script implementa un algoritmo genético para resolver un problema de optimización. En particular, se utiliza para encontrar la asignación óptima de productos a estantes en una tienda.

3. AnalisisProductos.py:

   Este script realiza un análisis de los productos vendidos en una tienda y su distribución en los estantes.

4. Distancia_Aestrella.py:

   Este script define una función para calcular la distancia entre dos puntos en un mapa utilizando el algoritmo de búsqueda A*.

5. Funciones_Genetico.py:

   Este script contiene las funciones necesarias para implementar el algoritmo genético en AlgoritmoGenetico.py.

6. GenTemperatura.py:

   Este script implementa un algoritmo para generar una temperatura inicial óptima para un algoritmo de enfriamiento simulado.

7. GeneracionProductos.py:

   Este script genera un conjunto de productos de forma aleatoria para su posterior distribución en los estantes de una tienda.

8. MapaSolucion.py:

   Este script muestra la solución óptima encontrada por el algoritmo genético para el problema de asignación de productos a estantes en una tienda.

9. Temple.py:

Este script implementa el algoritmo de enfriamiento simulado (también conocido como temple simulado) para resolver el problema de la mochila.

10. Temple2.py:

    Este script implementa el algoritmo de enfriamiento simulado para resolver el problema de asignación de productos a estantes en una tienda.

11. gen_Mapa.py:

    Este script genera un mapa de una tienda con la distribución de los estantes y asigna un identificador único a cada uno de ellos.

b. Diseño de los algoritmos

En relación al diseño de los algoritmos descritos en la seccióna anterior, se deciden implementarlos de la siguiente manera:

- En el script "A_estrella.py" se implementó el algoritmo de búsqueda A* para encontrar la ruta más corta en un mapa generado por el script "gen_Mapa.py". Se realizaron experimentos para medir el tiempo de ejecución y la longitud de la ruta encontrada.
- En el script "AlgoritmoGenético.py" se implementó un algoritmo genético para encontrar una combinación óptima de características de los productos generados por el script "GeneracionProductos.py". Se realizaron experimentos para medir el tiempo de ejecución y la calidad de la solución encontrada.
- En el script "Temple.py" se implementó el algoritmo de temple simulado para encontrar una solución óptima para el problema de asignación de productos a estantes en un mapa generado por el script "gen_Mapa.py". Se realizaron experimentos para medir el tiempo de ejecución y la calidad de la solución encontrada

En los anexos del presente informe se encuentran explicado en una tabla los detalles de cada uno de los scripts implementados, la relación que guardan entre ellos y el enlace al repositorio para poder conocer su estructura y código más en profundidad

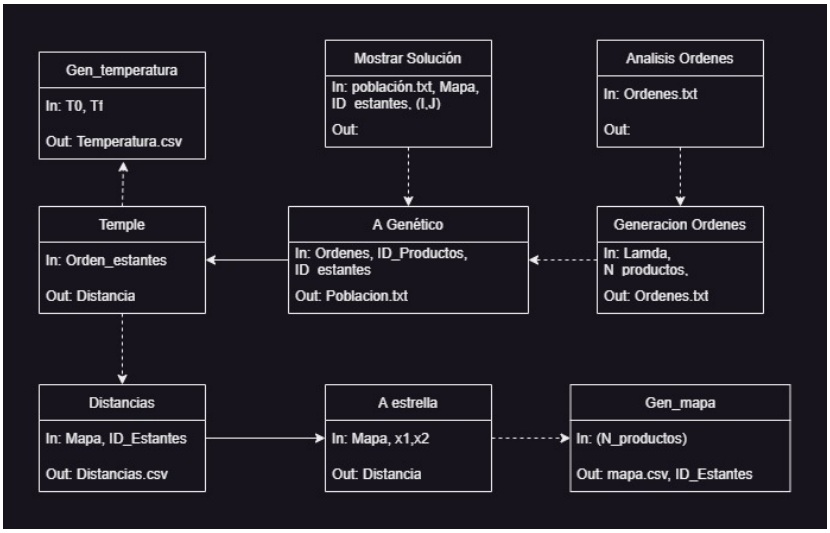
1. Estructuras de datos utilizadas

Al momento de describir las estructuras de datos utilizadas para el desarrollo de cada una de las consignas y algoritmos fue necesario implementar la siguiente estructura:

Script de Python	Estructuras de datos utilizadas
A_Estrella	Diccionarios, listas y tuplas para representar el grafo, y la información del algoritmo.
AlgoritmoGenetico	Listas y diccionarios para representar la

	población y la información del algoritmo.
AnalisisProducto	Data Frames de pandas para representar y manipular los datos de los productos.
Distancia_Aestrella	Listas y diccionarios para representar el grafo y la información del algoritmo.
Funciones_Genético	Listas y diccionarios para representar la población y la información del algoritmo.
gen_Mapa	Numpy arrays y dataframes de pandas para representar el mapa y la información de los estantes.
GenTemperatura	Listas para representar la temperatura y la información del algoritmo.
GeneracionProductos	Listas y diccionarios para representar los productos y la información del algoritmo.
MapaSolucion	Listas y diccionarios para representar la solución y la información del algoritmo.
Temple	Listas y diccionarios para representar la solución y la información del algoritmo.
Temple2	Listas y diccionarios para representar la solución y la información del algoritmo.

Cabe destacar que en la implementación de nuestra solución al problema presentado, no se realizó una estructura de desarrollo basado en el paradigma de programación orientada a objetos con la implementación de clases pero si se presenta a continuación un esquema de flujos que permite entender cómo se administra nuestra implementación:



c. Descripción de los experimentos realizados

1. Generación de casos de prueba

Teniendo en cuenta la implementación desarrollada

Para el script "GeneracionProductos.py" se generaron casos de prueba aleatorios utilizando distribuciones de probabilidad para las características de los productos.

En el script "gen_Mapa.py" se generó automáticamente un mapa de estantes y costos asociados.

2. Tamaño de los casos de prueba

En el script "GeneracionProductos.py" se generaron diferentes tamaños de casos de prueba, desde 10 hasta 100 productos distribuidos en diferentes ensayos entre 10 y 100 órdenes.

En el script "gen_Mapa.py" se generó un mapa con 256 estantes.

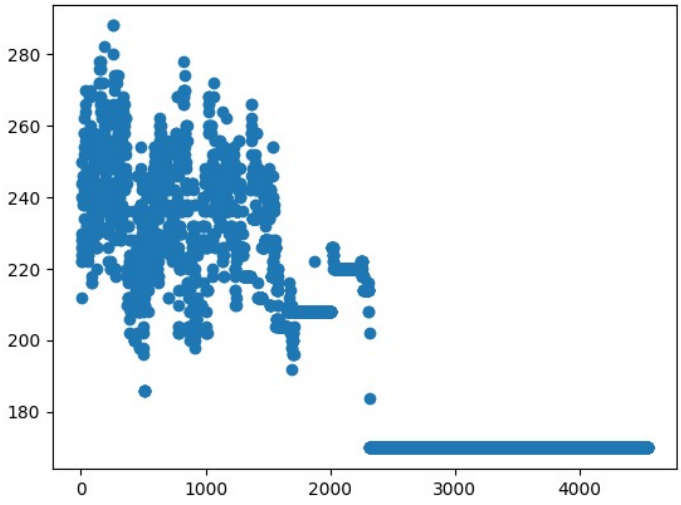
3. Metodología utilizada para los experimentos

En general, se utilizó un enfoque experimental basado en la comparación de los resultados obtenidos por cada uno de los algoritmos implementados en diferentes casos de prueba.

III. Resultados

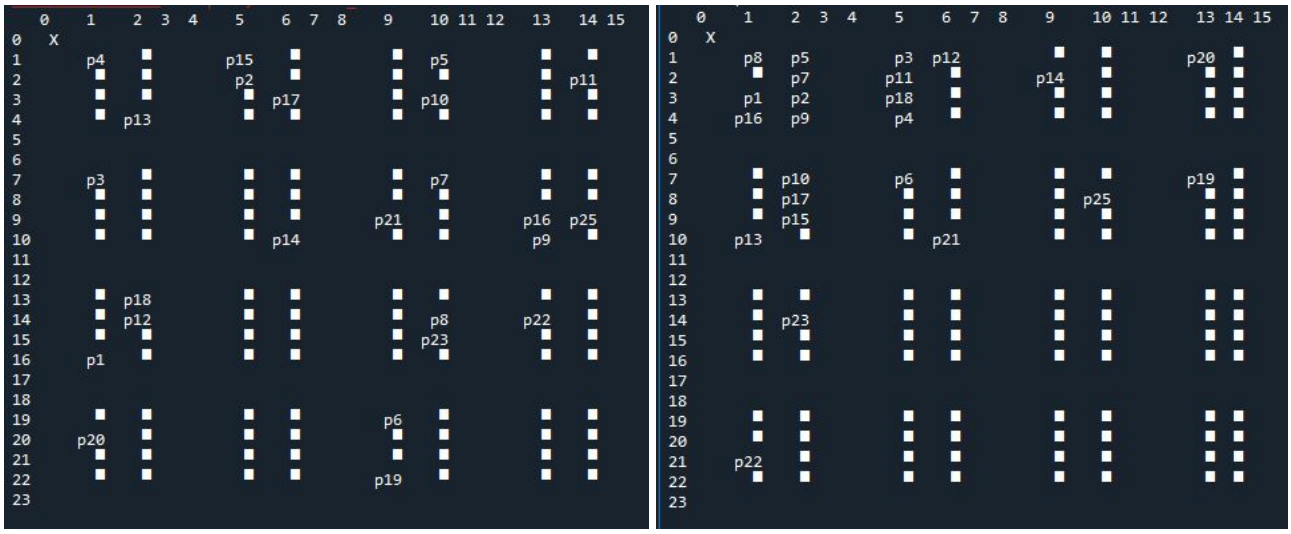
A continuación se presentan los resultados de la implementación de nuestra solución, siendo imprescindible presentar algunas gráficas que puedan corroborar el funcionamiento adecuado de la misma.

En una primera instancia se presenta el resultado de nuestro desarrollo del algoritmo de temple simulado en donde se puede apreciar como se logra la convergencia luego de una cierta cantidad de iteraciones del mismo. En este grafico se esta evaluando el costo del problema frente al número de iteraciones.



A modo de complemento también se presenta una representación de como el algoritmo genético se implementa para poder ordenar los 25 productos distribuidos en 100 ordenes de la manera más eficientes. En la primera imagen se presenta como se distribuyen los productos de dichas ordenes en la primera

iteración del mismo y en la segunda, se representa el ordenamiento que generó nuestro algoritmo luego de 500 iteraciones



Cómo se puede visualizar, hay un ordenamiento de los productos de forma tal que cómo el ingreso se presenta en donde esta la X en las figuras, nuestra solución ubica los productos de las ordenes de manera que la distancia entre ellos sea la óptima para las 100 ordenes que se tienen de los 25 productos.

IV. Conclusiones

a. Conclusiones generales

Los algoritmos implementados (A*, algoritmo genético y temple simulado) tienen distintas fortalezas y debilidades en términos de eficiencia y calidad de la solución, dependiendo del problema a resolver.

Los resultados obtenidos en los experimentos muestran que los algoritmos genéticos son los más efectivos para encontrar soluciones óptimas en términos de tiempo y calidad de la solución para problemas de optimización combinacional, mientras que A* es más efectivo para problemas de búsqueda en grafos y temple simulado puede ser utilizado para problemas de optimización continua.

La generación de casos de prueba es un aspecto crítico en la evaluación de los algoritmos y se utilizaron distintas metodologías para generar casos de prueba de distintos tamaños y características.

Los resultados obtenidos son prometedores y sugieren que los algoritmos implementados tienen un gran potencial para ser utilizados en aplicaciones prácticas.

b. Limitaciones del proyecto

La implementación de los algoritmos podría ser optimizada para mejorar su eficiencia y escalabilidad, especialmente para problemas de mayor tamaño y complejidad.

La evaluación de los algoritmos se limitó a un conjunto de casos de prueba generados de forma aleatoria, por lo que los resultados podrían variar en función de las características del problema real.

No se consideraron otros algoritmos o técnicas de optimización que podrían ser igualmente efectivos o incluso superiores para resolver los problemas específicos abordados en este proyecto.

El análisis de los resultados podría ser mejorado utilizando técnicas estadísticas más sofisticadas, como análisis de varianza o pruebas de hipótesis.

c. Trabajos futuros

1. **Optimización de los algoritmos:**

   Para el algoritmo de A* (A_estrella.py) se podría probar con distintas heurísticas, con el objetivo de mejorar su rendimiento. También podría explorarse la implementación de una versión paralela del algoritmo para aprovechar mejor los recursos del equipo.

   En el caso del algoritmo genético (AlgoritmoGenético.py), se podría probar con distintas configuraciones de parámetros y estrategias de selección de individuos para mejorar su capacidad de convergencia.

   Para el temple simulado (Temple.py y Temple2.py), se podría explorar la implementación de una versión paralela que permita explorar con mayor eficiencia el espacio de soluciones.

2. **Incorporación de nuevas funcionalidades:**

   Se podría agregar una funcionalidad para que el usuario pueda ingresar su propio mapa de estantes, en lugar de generar uno aleatorio como se hace actualmente.

   También se podría implementar una funcionalidad para que el usuario pueda seleccionar la cantidad de estantes que desea que haya en el mapa, en lugar de fijarse en 100 como se hace actualmente.

   Otra posible funcionalidad sería la de permitir que el usuario pueda definir una lista de productos que desea ubicar en los estantes, en lugar de generar una lista aleatoria como se hace actualmente.

3. **Mejoras en la presentación de resultados:**

   Se podría implementar una visualización gráfica del mapa de estantes y de la solución encontrada por los distintos algoritmos, para facilitar la comprensión de los resultados.

   También se podrían generar estadísticas sobre el desempeño de los distintos algoritmos en función del tamaño del problema, y presentarlas en forma de gráficos o tablas para facilitar su análisis.

4. **Mejoras en la documentación:**

   Se podría mejorar la documentación de los distintos scripts, incorporando descripciones más detalladas de las funciones y

estructuras de datos utilizadas, así como también diagramas de flujo y de clase para facilitar su comprensión.

También se podría incorporar una sección de "troubleshooting" en la documentación, para que los usuarios puedan resolver posibles problemas que puedan surgir al utilizar el programa.

Anexos

a. Descripción detallada de los códigos de los programas desarrollados

Nombre	Consigna	Descripción	Enlace
Códigos de Python			
A_estrella.py	1	El código es una implementación del algoritmo A* para buscar la ruta más corta entre un punto de inicio y un punto de destino en un mapa. La función heuristica calcula la distancia de Manhattan entre dos puntos, lo que se usa para estimar la distancia entre el punto actual y el punto final. La función A_estrella toma un mapa, un diccionario de costos, la posición de inicio y la posición de destino como entrada y devuelve una lista de posiciones que conforman la ruta más corta. El algoritmo A* se basa en una función de evaluación f(x) que tiene en cuenta tanto el costo acumulado hasta un punto x como una estimación heurística del costo restante para llegar al destino. El algoritmo intenta minimizar esta función de evaluación al elegir el siguiente punto a explorar en cada iteración. El algoritmo utiliza tres listas para llevar un seguimiento de los nodos abiertos, los nodos cerrados y los nodos solución. La lista de nodos abiertos contiene los nodos que aún no se han explorado. La lista de nodos cerrados contiene los nodos que ya se han explorado y cuyo costo total se ha calculado. La lista de nodos solución contiene la ruta más corta desde el punto de inicio hasta el punto de destino. En cada iteración del algoritmo, se explora el nodo actual y se generan los nodos vecinos. Cada vecino se evalúa y se agrega a la lista de nodos abiertos si aún no se ha explorado. Luego se elige el nodo con la evaluación más baja en la lista de nodos abiertos y se repite el proceso hasta llegar al nodo final.	https://github.com/matiasLinaresG/IA-II/blob/main/TP1/A_estrella.py

AlgoritmoGeneica.py	3	El código proporcionado es un algoritmo genético que resuelve un problema de optimización. El objetivo del algoritmo es encontrar la mejor solución (mejor combinación de elementos) para cumplir con una serie de órdenes. Para ello, el algoritmo utiliza una población inicial de soluciones aleatorias y luego va mejorando esta población iterativamente. En cada iteración, el algoritmo evalúa la calidad de cada solución en la población utilizando una función de calidad. Luego, selecciona un subconjunto de las soluciones más aptas (en términos de calidad) para reproducirse y crear nuevas soluciones. Estas nuevas soluciones se generan mediante un proceso de cruce y mutación, que toma dos soluciones existentes y crea nuevas soluciones combinando partes de ambas. Este proceso se repite hasta que se alcanza una solución óptima o se llega al límite de iteraciones. El algoritmo se ejecuta utilizando múltiples procesadores, lo que acelera el proceso de cálculo. Además, el algoritmo incluye opciones para leer y escribir archivos de texto que contienen información sobre la población de soluciones y las órdenes a procesar. El código importa una serie de funciones desde un archivo de Python llamado "Funciones_Genetico.py". Estas funciones contienen la lógica principal del algoritmo y son responsables de calcular la calidad de una solución, seleccionar soluciones para reproducirse y crear nuevas soluciones a partir de soluciones existentes. El código también utiliza las bibliotecas estándar de Python "numpy" y "pandas" para realizar operaciones matemáticas y manejar datos estructurados, respectivamente.	https://github.com/matiasLinaresG/IA-II/blob/Cambio-de-salida/TP1/AlgoritmoGenetico.py
AnalisisProducto.py	3	Este código analiza los productos de un conjunto de órdenes almacenadas en un archivo de texto llamado "ordenes_mod.txt". Primero, lee el contenido del archivo y divide las órdenes en una lista llamada "ordenes". Luego, elimina el primer elemento de cada lista de órdenes y las órdenes vacías. Posteriormente, guarda las órdenes en un nuevo archivo de	https://github.com/matiasLinaresG/IA-II/blob/Cambio-de-salida/TP1/AnalisisProduct

		texto llamado "ordenes_ordenadas.txt", separando los productos por comas. A continuación, crea una lista de todos los productos ordenados sin repetición y los guarda en un archivo CSV llamado "productos.csv" utilizando la biblioteca pandas. Luego, cuenta cuántas veces se repite cada producto en las órdenes y los guarda en una lista llamada "productos". Finalmente, grafica en dos ventanas diferentes: 1. La cantidad de veces que se repite cada producto, ordenados de mayor a menor. 2. La cantidad de productos por orden. Cabe destacar que hay una línea comentada que solicita al usuario una entrada para que la ejecución del programa se detenga hasta que el usuario presione una tecla.	os.py
Distancia_Aestrella.py	1	Este código en Python utiliza la librería pandas y numpy para leer archivos CSV y almacenar datos en un dataframe. Además, utiliza una función A_estrella importada de un archivo A_estrella.py para calcular la distancia entre pares de puntos en un mapa. Primero, se lee un archivo mapa.csv que contiene un mapa en forma de matriz y otro archivo costo.csv que contiene los costos asociados a cada elemento de la matriz. Luego, se lee otro archivo ID_estantes.csv que contiene información sobre la ubicación de los almacenes en el mapa. A continuación, se agrega una nueva fila en el dataframe IdAlmacen con un nuevo almacén ubicado en (0,0) y se guarda el dataframe en el archivo ID_estantes.csv. Después, se calcula la distancia entre cada par de almacenes utilizando la función A_estrella y se guarda el resultado en un dataframe Distancias. Finalmente, se muestra el dataframe Distancias y se guarda en un archivo	https://github.com/matiasLinaresG/IA-II/blob/Cambio-de-salida/TP1/Distancia_Aestrella.py

		distancias.csv.	
Funciones_ Genetico.py	3	El archivo "Funciones_Genetico.py" contiene funciones necesarias para el algoritmo genético que resuelve el problema de ordenamiento de estantes en un depósito para optimizar el tiempo de recogida de productos. Las funciones en este archivo incluyen: - `**DeProductosAEstantes(orden, IdEstantes, configuracion)**`: Esta función toma una lista de productos en un orden y devuelve una lista de estantes donde se encuentra cada producto en la configuración actual de los estantes. `IdEstantes` es una lista que contiene los ID de cada estante, mientras que `configuracion` es una lista que contiene los productos en el orden en que aparecen en cada estante. - `**Calidad(Poblacion_P, Ordenes_P, IdEstantes)**`: Esta función calcula la calidad de la población proporcionada, donde cada individuo de la población es una configuración de los estantes. Utiliza la función `Calidad_individual` para calcular la calidad individual de cada configuración y devuelve una lista de la calidad de cada individuo en la población. - `**Calidad_individual(Individuo, Ordenes_P, IdEstantes)**`: Esta función calcula la calidad de un solo individuo, es decir, la cantidad de tiempo que tarda en cumplir todas las órdenes de los clientes. Utiliza la función `DeProductosAEstantes` para obtener la lista de estantes donde se encuentra cada producto en el orden de cada orden de cliente. Luego utiliza la función `temple_simulado` para obtener el tiempo de recogida de cada orden de cliente y devuelve el promedio de estos tiempos como la calidad del individuo. - `**Cruce(padre1, padre2)**`: Esta función realiza el cruce de dos padres y devuelve dos hijos. Se selecciona aleatoriamente un punto de cruce y se intercambian los genes en los padres desde ese punto. Los hijos resultantes contienen todos los genes de ambos padres sin duplicados, a menos que el gen "vacio" aparezca en ambos padres, en cuyo caso se mantiene solo una copia. - `**Mutacion(Individuo, Cantidad)**`: Esta función realiza la mutación en un solo individuo al intercambiar aleatoriamente los genes en dos puntos. La cantidad de intercambios se determina aleatoriamente a partir de un valor de entrada.	https://github.com/matiasLinaresG/IA-II/blob/Cambio-de-salida/TP1/Funciones_Genetico.py

		- `**Convergencia(calidades, N)`:** Esta función determina si la población ha convergido al encontrar un número de individuos con la misma calidad (es decir, el mismo tiempo de recogida de productos). El número mínimo de individuos necesarios para considerar que la población ha convergido se determina a partir de un valor de entrada `N`. - `**SelecionarPoblacion(Poblacion, calidad, Cantidad)`:** Esta función selecciona los mejores individuos de una población en función de su calidad. Devuelve las `Cantidad` mejores calidades y configuraciones de estantes.	
GenTemperatura.py	2	La función "GenTemperatura" genera una lista de descenso de temperatura en un archivo CSV que puede ser utilizado en un programa de temple simulado. La temperatura inicial es de 50 grados y la temperatura final es de 0.1 grados. La función utiliza la fórmula de descenso de temperatura "temp_actual = temp_actual / math.log(2.722)" para calcular los valores de temperatura en cada iteración. Los valores de temperatura y número de iteración se almacenan en dos listas separadas y se utilizan para crear un dataframe de Pandas que se guarda en un archivo CSV llamado "Temperatura.csv". El número total de valores de temperatura generados se imprime en la consola al final del proceso.	https://github.com/matiasLinaresG/IA-II/blob/Cambio-de-salida/TP1/GenTemperatura.py
GeneracionProductos.py		Este código genera una lista de órdenes de compra de productos. El número de órdenes y el número máximo de productos por orden se definen al principio del código. El código utiliza la distribución geométrica para determinar la cantidad de productos que se deben comprar en cada orden. Luego, se utiliza otra distribución geométrica para generar números de productos aleatorios. El código también verifica que no se generen números de productos duplicados o fuera del rango permitido. Finalmente, el código crea un archivo de texto con las órdenes de compra generadas. Cada orden se identifica con un número y se lista los productos que se deben comprar.	https://github.com/matiasLinaresG/IA-II/blob/Cambio-de-salida/TP1/GeneracionProductos.py

MapaSolucion.py	3	Este código parece estar diseñado para generar un mapa de solución para un problema específico, utilizando una población de individuos previamente generados y evaluados por una función de calidad individual (definida en un módulo llamado "Funciones_Genetico"). El objetivo es encontrar la mejor solución a un problema de almacenamiento, donde se deben organizar productos en estantes de un almacén de forma tal que se cumplan ciertas restricciones dadas por una lista de órdenes de productos a ser retirados en cierto orden. El código comienza leyendo la población previamente generada y almacenada en un archivo llamado "poblacion.txt", así como la información de los estantes y los productos disponibles en el almacén. También se lee el archivo "ordenes_ordenadas.txt", que contiene la lista de órdenes de productos. Luego, el código utiliza multiprocessing para evaluar la calidad de cada individuo en la población utilizando la función Calidad_individual definida en el módulo "Funciones_Genetico". La evaluación se realiza en paralelo para acelerar el proceso. A continuación, el código genera un mapa de solución a partir del individuo de mayor calidad en la población, organizando los productos en los estantes del almacén de acuerdo a su posición en el individuo. Por último, se muestra el mapa resultante utilizando la biblioteca pandas. Es importante destacar que el código parece estar diseñado para ser utilizado en conjunto con otros módulos y archivos de entrada específicos, por lo que su uso sin el contexto adecuado podría resultar en errores.	https://github.com/matiasLinaresG/IA-II/blob/Cambio-de-salida/TP1/MapaSolucion.py
Temple2.py	2	Este código implementa el algoritmo de Temple Simulado para resolver el problema de picking. El algoritmo comienza importando las librerías necesarias, incluyendo la distribución de Boltzmann. A continuación, se define la función `temple_simulado`, que toma como entrada una lista de estantes y devuelve una lista de estantes reordenados que representan una solución al	https://github.com/matiasLinaresG/IA-II/blob/Cambio-de-salida/TP1/T

		problema. Dentro de la función, se abre el archivo `distancias.csv` y se guarda su contenido en un diccionario. Se hace lo mismo con el archivo `Temperatura.csv`. Luego, se establece la temperatura inicial como el valor de la primera fila del archivo `Temperatura.csv`. Después se define una lista vacía para almacenar la solución y se establece el costo anterior en un valor alto. La lista de estantes se modifica añadiendo la bahía de carga con el id "carga" al principio y al final de la lista. A continuación, el código entra en un bucle que ejecuta la iteración del Temple Simulado. En cada iteración se seleccionan dos elementos de la lista de estantes al azar y se intercambian sus posiciones. Después se calcula el costo de la solución actual recorriendo la lista de estantes y sumando la distancia entre cada par de estantes consecutivos utilizando el diccionario de distancias. Si la solución actual es mejor que la solución anterior, se acepta. Si no es mejor, se acepta con una probabilidad dada por la distribución de Boltzmann. El valor de la temperatura se actualiza en cada iteración y el bucle continúa hasta que la temperatura llega a un valor mínimo o se alcanza un número máximo de iteraciones. Por último, el código devuelve la solución encontrada en forma de lista de estantes reordenados.	emple2.py
gen_Mapa.py	1	El código "gen_Mapa.py" contiene la clase "Mapa" que se encarga de generar un mapa para una bodega, en el cual se distribuyen estantes. La matriz del mapa se genera superponiendo una celda unitaria que contiene ocho estantes. También se genera una matriz de costo que asigna un valor de costo a cada estante. Además, se asignan IDs a cada estante en una lista de posiciones de estantes, que se almacena en un archivo CSV.	https://github.com/matiasLinaresG/IA-II/blob/Cambio-de-salida/TP1/gen_Mapa.py

		La clase Mapa tiene un constructor que toma un número de estantes como argumento. Si no se proporciona un número de estantes, se asigna un valor predeterminado de 20. La matriz de mapa y la matriz de costo se generan automáticamente en el constructor. El método "asignar_ids" asigna un ID a cada estante y los almacena en una lista de posiciones de estantes. La lista se convierte en un DataFrame y se almacena en un archivo CSV. El código utiliza la biblioteca pandas para manejar los DataFrames y los archivos CSV. La matriz de mapa y la matriz de costo también se almacenan en archivos CSV.	

b. Archivos csv y txt empleados durante la implementación de la solución

Archivos CSV	Archivos de texto
ID_estantes.csv	ordenes_mod.txt
costo.csv	ordenes_ordenadas.txt
distancias.csv	población.txt
mapa.csv	
productos.csv	