UNCUYO
UNIVERSIDAD NACIONAL DE CUYO

FACULTAD DE INGENIERÍA

TRABAJO FINAL

VISION ARTIFICIAL

INTELIGENCIA ARTIFICIAL I

Alumno: Vignolo Alan

Legajo: 12667

Profesora: Dra. Ing. Selva Soledad Rivera

Índice

Resumen

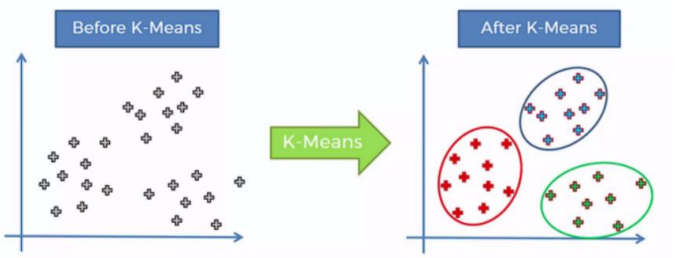
Este proyecto aborda el desafío de identificar y reorganizar el contenido de cuatro cajas apiladas utilizando visión artificial y métodos de clasificación de imagenes. Las cajas contienen tornillos, tuercas, clavos o arandelas, y su orden de apilamiento debe ser ajustado. Implementamos algoritmos K-means y K-nn para la clasificación de los objetos identificados por la visión artificial y los comparamos para determinar la opción más aproximada. También utilizamos el lenguaje STRIPS para desarrollar un plan que permita al robot reordenar las cajas en una configuración deseada. Una vez reordenadas, el robot tiene que transportar las cajas de un punto A un punto B, para lo cual implementamos el algoritmo de búsqueda A* con una heurística de Distancia de Manhattan para trazar el camino más corto. Los resultados obtenidos son analizados y discutidos en detalle.

Introducción

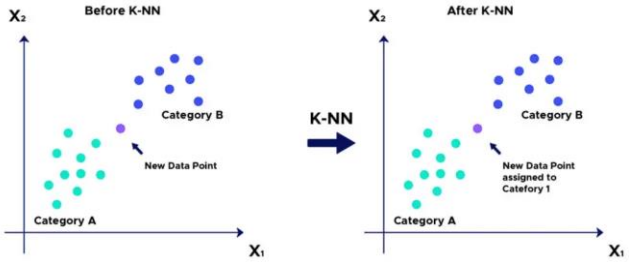
En el vasto mundo de la Inteligencia Artificial, encontramos una serie de técnicas y algoritmos que abordan distintos desafíos, desde la interpretación de imágenes hasta la navegación en entornos complejos. En el proyecto en cuestión, se requiere del desarrollo de una solución que integre cuatro componentes esenciales: la visión artificial, la clasificación de objetos, la reorganización de dichos objetos y, por último, su transporte a través de un entorno laberíntico.

La visión artificial, una disciplina de la Inteligencia Artificial, nos permite emular la capacidad humana de interpretar y analizar imágenes, convirtiéndola en una herramienta ideal para identificar y clasificar los objetos contenidos en las cajas de nuestro problema. Sin embargo, la correcta identificación y clasificación requiere del uso de algoritmos de aprendizaje eficientes y efectivos. Aquí es donde entran en juego los algoritmos K-means y K-nn.

El algoritmo K-means, un método de agrupamiento no supervisado, es ampliamente utilizado en el análisis de datos para la clasificación y la identificación de patrones. Este algoritmo tiene la ventaja de ser simple pero potente, permitiendo clasificar rápidamente grandes volúmenes de datos en grupos o "clusters" basados en sus características.



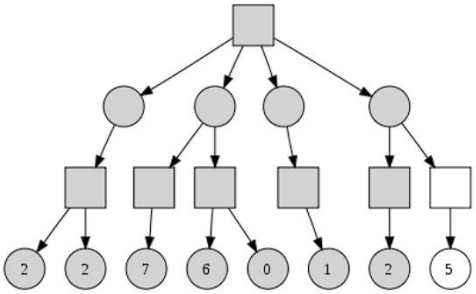
Por otro lado, tenemos el algoritmo K-nn o K-Nearest Neighbors, un método de clasificación supervisado, que clasifica los elementos en base a la proximidad a los vecinos más cercanos en el espacio de características. K-nn es altamente adaptable y puede utilizarse en un amplio rango de aplicaciones, siendo particularmente útil cuando los patrones a clasificar no son linealmente separables.

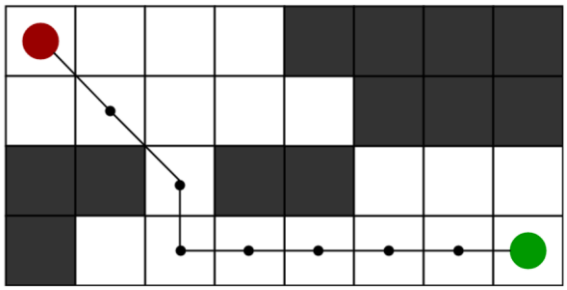


Una vez identificados y clasificados los objetos, necesitamos reorganizar las cajas. Aquí es donde entra en escena el lenguaje STRIPS (Stanford Research Institute Problem Solver). STRIPS es un lenguaje formal para la planificación de acciones, es decir, permite describir un estado inicial, un estado objetivo y un conjunto de acciones posibles. Utilizando STRIPS,

podemos determinar una secuencia de acciones que nos lleve del estado inicial al estado objetivo, proporcionando una solución para la reorganización de las cajas.

Por último, para transportar las cajas desde un punto A hasta un punto B a través de un laberinto, recurrimos al algoritmo A*. Este es un algoritmo de búsqueda informada que utiliza una heurística, en nuestro caso la Distancia de Manhattan, para encontrar el camino más corto en un mapa o gráfico. A* es ampliamente conocido por su eficiencia y precisión, siendo una elección popular para problemas de rutas y navegación.





Especificación del Agente

El agente en este caso es un robot equipado con visión artificial y otros sensores y actuadores necesarios para realizar tareas específicas. Está diseñado para identificar el contenido de cuatro cajas apiladas, reorganizarlas en una nueva configuración, y guiar al robot a través de un laberinto hasta un destino específico.

Las características del agente se describen en la siguiente tabla REAS (Rendimiento, Entorno, Actuadores, Sensores):

	Descripción
Rendimiento	Identificar el contenido de cuatro cajas apiladas y reorganizarlas en una nueva configuración. Guiar al robot a través de un laberinto hasta un destino específico por el camino más corto.
Entorno	Un entorno con cuatro cajas apiladas y un laberinto con obstáculos.
Actuadores	Brazos del robot para manipular las cajas y ruedas motorizadas para moverse a través del laberinto.
Sensores	Cámaras para tomar imágenes de las cajas y del laberinto, así como sensores de distancia y otros sensores necesarios para detectar obstáculos y evitar colisiones.

Las propiedades del entorno de trabajo se describen a continuación:

Propiedad	**Descripción**
Total, o parcialmente observable	Totalmente observable. El robot puede ver las cajas y el laberinto desde su posición actual y tiene acceso a la información completa del entorno.
Determinista o estocástico	Determinista. El robot siempre realizará las mismas acciones en una situación dada.
Episódico o secuencial	Episódico. El robot se enfrentará a cada tarea (identificar el contenido de las cajas y guiar al robot a través del laberinto) como un episodio independiente.
Estático o dinámico	Estático. El contenido de las cajas y la disposición del laberinto no cambian durante la ejecución de la tarea.
Discreto o continuo	Discreto. Hacemos una simplificación del entorno ya que tenemos en cuenta un numero entero y finito de cajas, así como dividir el laberinto en cuadrados.
Agente individual o multiagente	Agente individual. El robot es el único agente en el entorno de trabajo.

Considerando las características del proyecto, nuestro agente es un robot que se clasifica como un agente que aprende. Utilizando algoritmos K-means y K-nn, el agente es capaz de identificar tornillos, clavos, tuercas y arandelas en cuatro cajas apiladas, reflejando así su capacidad de aprendizaje. Por otro lado se podría considerar un agente basado en objetivos ya que con el algoritmo de A*, este decide que hacer para encontrar una secuencia de acciones que conduzcan a un estado deseable.

Diseño del agente

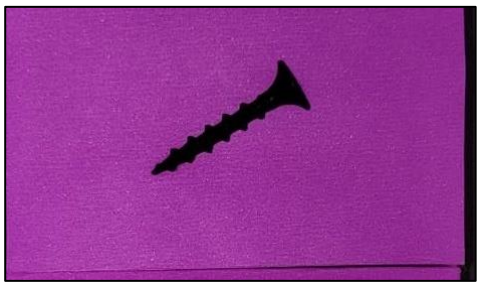
Para el desarrollo de este agente se decidió dividir el problema en 3, algoritmo de visión artificial para el reconocimiento de piezas, algoritmo que mediante lenguaje Strips encuentra los pasos para un nuevo arreglo y algoritmo A* para encontrar el camino entre dos puntos.

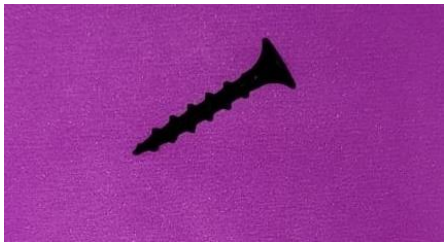
Visión Artificial

En este punto comenzaremos con el preprocesamiento de las imágenes de entrenamiento para lograr que los algoritmos de asignación logren identificar con facilidad la clase perteneciente de una nueva imagen. Para esto se aplican varios filtros y en una primera instancia se probó con 2 tipos de extractores de características: HOG y Momentos de HU.

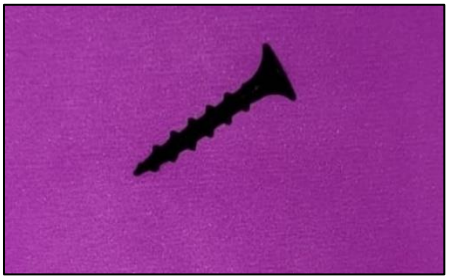
1. Preprocesamiento de imágenes y extracción de características

Lectura y recorte de la imagen: Se empieza por abrir la imagen que se desea procesar. Una vez abierta, se realiza un recorte a los bordes de la imagen para eliminar cualquier tipo de ruido o interferencia no deseada que pueda afectar el análisis posterior. Este recorte contribuye a la limpieza inicial de la imagen, asegurando que solo los elementos de interés permanezcan en la imagen.

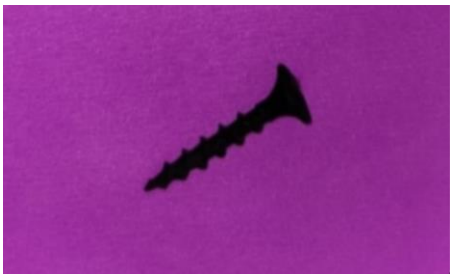




Redimensionamiento: La imagen es redimensionada a un tamaño estándar. Esta estandarización es importante porque permite un procesamiento de imágenes más uniforme. Al tener todas las imágenes del mismo tamaño, garantizamos que las operaciones aplicadas posteriormente se comporten de la misma manera en todas las imágenes.

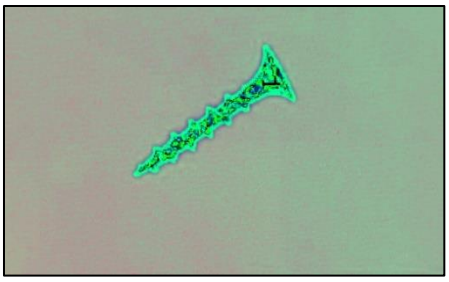


Aplicación de filtro gaussiano: Se aplica un filtro de suavizado a la imagen para reducir su ruido y detalles menores, dejando solo las características más prominentes. Este filtro es llamado gaussiano debido a que los valores utilizados para el suavizado siguen una distribución gaussiana, lo que permite un promedio ponderado de los píxeles de la imagen, dando mayor importancia a los píxeles cercanos al centro del kernel de suavizado.

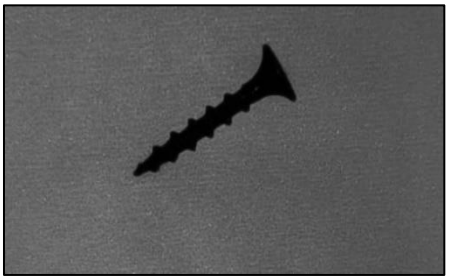


Conversión a HSV de imagen original y Segmentación por color:

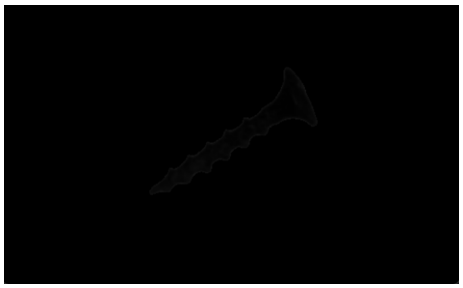
- Conversión a HSV: Cada pixel en una imagen digital tiene un valor de color representado en el espacio de color RGB (rojo, verde, azul). Sin embargo, en el espacio de color HSV (matiz, saturación, valor), el color de un píxel se define por su tono, no por su combinación de colores primarios. Durante la conversión, los valores RGB de cada píxel se mapean a los valores correspondientes de HSV utilizando transformaciones matemáticas.
- Segmentación por color: Luego de la conversión, se aplica un umbral a los píxeles de la imagen basándose en los valores de HSV. Cada píxel de la imagen se evalúa para determinar si su valor de HSV se encuentra dentro del rango definido. Si es así, el píxel se mantiene; si no, se descarta (generalmente cambiándolo a negro).



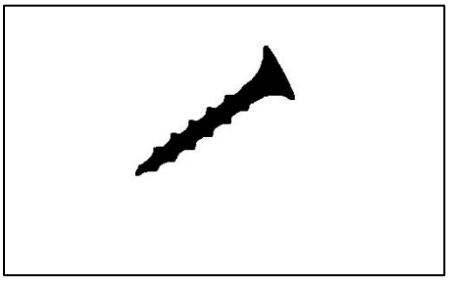
Conversión a escala de grises: Las imágenes en color, aunque contienen mucha información, pueden ser complejas para procesar debido a los múltiples canales de color. Para simplificar el análisis, se convierte la imagen a escala de grises, donde cada píxel tiene un solo valor que representa su intensidad de luz. Esto reduce significativamente la complejidad de la imagen y permite un procesamiento más eficiente.



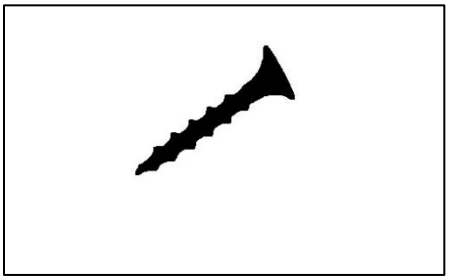
Aplicación de la máscara a la imagen en escala de grises: Después de la segmentación, obtenemos una máscara, que es una imagen binaria que contiene píxeles blancos donde los objetos de interés están presentes y píxeles negros en todos los demás lugares. Al aplicar la máscara a la imagen, se realiza una operación lógica AND en cada píxel. Si el píxel de la máscara es blanco, el píxel de la imagen original se mantiene; si el píxel de la máscara es negro, el píxel de la imagen original se convierte en negro.



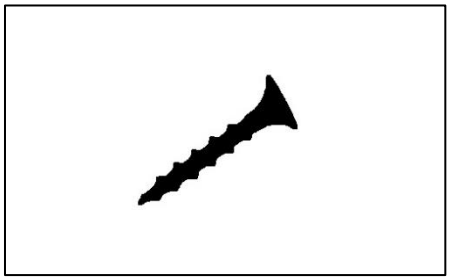
Detección de bordes: Luego de la aplicación de la mascara a la imagen, se realiza una detección de los bordes de la pieza. Se coloca el borde con el área mas grande en una imagen en blanco.



Dilatación y erosión: Estas operaciones morfológicas ayudan a suavizar los contornos del objeto en la imagen. La dilatación hace que el objeto se expanda, añadiendo píxeles al contorno exterior del objeto. La erosión, por otro lado, hace que el objeto se encoja, quitando píxeles del contorno exterior. El resultado de estas operaciones es una imagen con objetos de formas más suaves y uniformes.



Cálculo de centroides del objeto y traslación: Una vez que los objetos se han aislado y se han suavizado sus formas, se calcula el centroide, o el centro de su masa. El cálculo del centroide se realiza sumando las coordenadas (x, y) de todos los píxeles negros en la imagen y luego dividiendo por la cantidad total de estos píxeles. Esto te da el punto central de los píxeles negros, que es el centroide del objeto. La traslación se realiza ajustando las coordenadas (x, y) de cada píxel en la imagen. Esto se hace sumando o restando la cantidad necesaria para mover el centroide del objeto al centro de la imagen.



2. Extracción de características

En este apartado se realizaron 2 tipo de caracterización de las imágenes para comprobar cuál es la más efectiva en el trabajo a desarrollar. Estos son:

Histograma de Gradientes Orientados (HOG): El HOG es un descriptor de características que captura la forma y estructura de los objetos en la imagen al contabilizar las apariciones de direcciones de gradientes. En este proceso, la imagen se divide en celdas pequeñas y conectadas de 8x8 píxeles cada una. En cada una de estas celdas, se calcula un histograma de las orientaciones de gradiente utilizando nueve diferentes orientaciones. Posteriormente, estas celdas se agrupan en bloques más grandes (de 2x2 celdas) y los histogramas en cada bloque se normalizan para disminuir los efectos de las variaciones de iluminación. La aplicación de la normalización L2 y la transformación de raíz cuadrada previa al cálculo de los gradientes facilita la gestión de los contrastes en la imagen. En este caso se utiliza caracterizador luego de pasar la imagen a escala de grises, sin aplicar el resto del preprocesamiento de imagen.

Momentos de Hu: Los momentos de Hu son un conjunto de siete valores derivados de los momentos básicos de la imagen, que son medidas estadísticas de la distribución de los píxeles en la imagen. Los momentos de Hu son calculados de manera tal que son invariables a la escala, posición y orientación de la imagen. Esto significa que, sin importar cuánto se modifique

la posición, tamaño o rotación de una forma en una imagen, sus momentos de Hu permanecerán constantes. Estas propiedades hacen que los momentos de Hu sean muy útiles para el reconocimiento de formas y patrones en imágenes. En el código, estos momentos se calculan luego realizar todo el preprocesamiento de las imágenes, justo después de trasladar el objeto al centro de la imagen.

3. Reducción de dimensionalidad

Aplicamos el Análisis de Componentes Principales (PCA) para reducir la dimensionalidad de nuestro conjunto de datos para poder hacer una visualización lo más precisa posible en una imagen en 3D donde se ven claramente los valores de las caracterizaciones para cada imagen, los clústeres creados y los centroides según sea el caso. La idea principal de PCA es encontrar un nuevo conjunto de ejes ortogonales (los componentes principales) a lo largo de los cuales los datos tienen la máxima varianza.

4. Clasificación

Una vez extraídas las características, procedemos a la etapa de clasificación. Usamos dos técnicas: K-Means y K-Nearest Neighbors (KNN).

Aplicación del K-Means:

Dentro del contexto de este proyecto, se utilizó el algoritmo de K-Means, un método de aprendizaje no supervisado, para segmentar los datos. Este algoritmo tiene como objetivo principal dividir los datos en varios grupos o "clusters". En este caso, el número de clusters (K) se eligió en función del conocimiento previo que se tenía sobre la cantidad de clases distintas en el conjunto de datos.

Los pasos que seguimos para implementar K-Means fueron los siguientes:

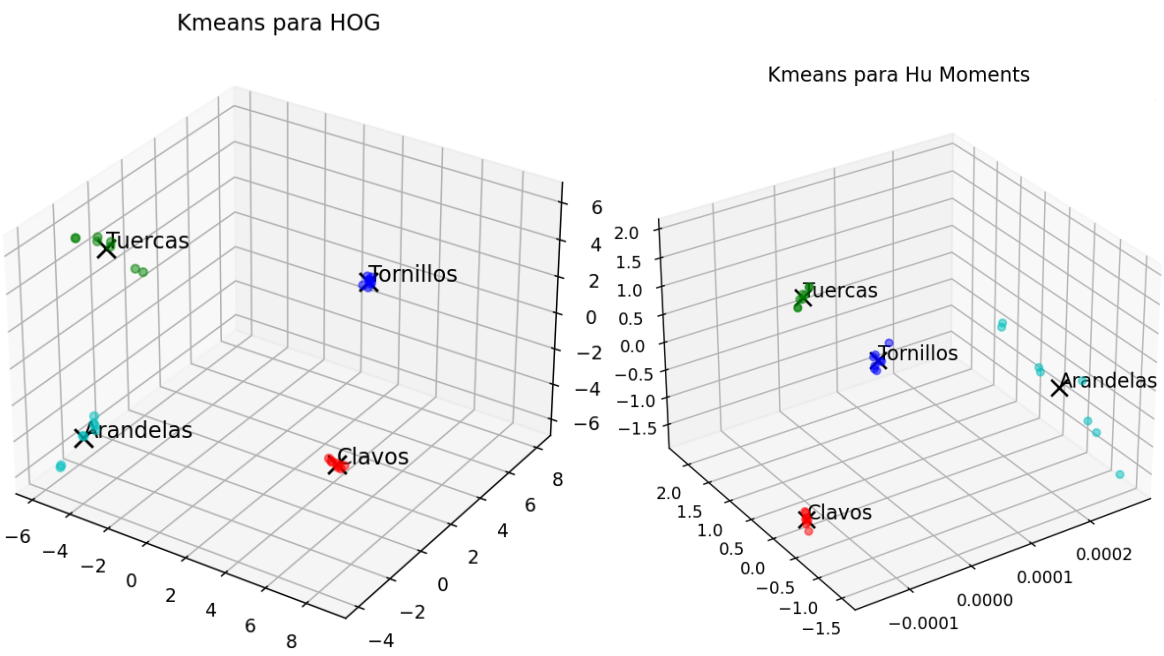
Primero, se seleccionaron 4 puntos aleatorios en el espacio de características como centros iniciales de los clusters.

Posteriormente, para cada punto de los datos, se calculó su distancia a cada centro de los clusters y asignamos cada punto al cluster más cercano.

Una vez asignados todos los puntos a los clusters, se recalculó los centros de los clusters como el promedio de todos los puntos que pertenecen a cada cluster.

Se repitieron los pasos 2 y 3 hasta que el valor de los centroides no varió de forma significativa.

Luego la predicción se hace en base a la distancia que tiene el conjunto de valores de caracterización de la imagen nueva a cada uno de los centroides obtenidos.



5. Implementación del K-Nearest Neighbors (KNN):

Como segunda opción, utilizamos KNN, un algoritmo de aprendizaje supervisado. Este algoritmo asigna un objeto a la clase más común entre sus K vecinos más cercanos.

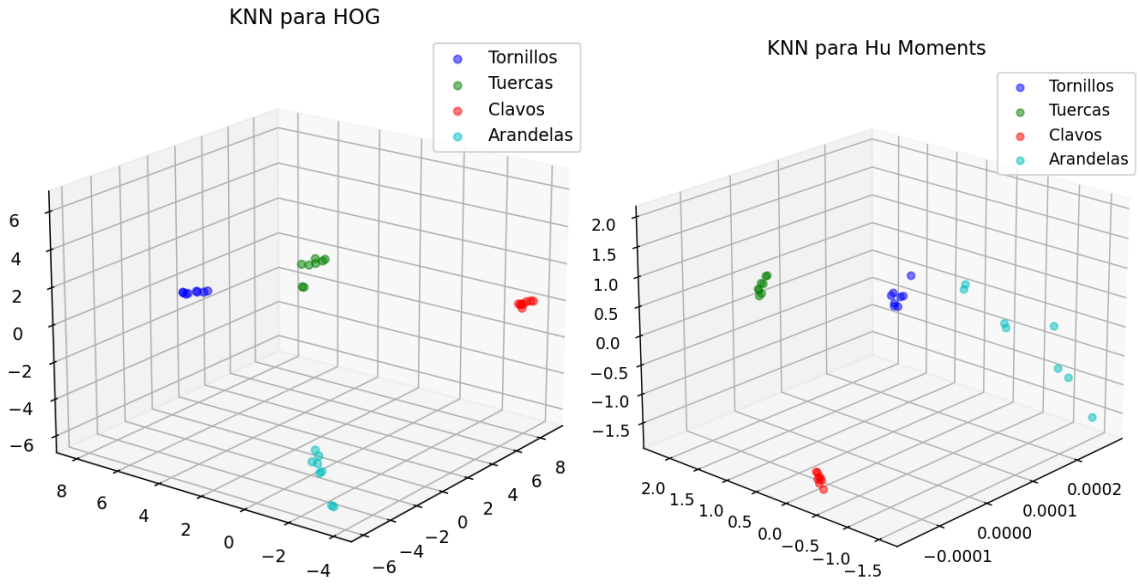
Para implementar KNN, seguimos los siguientes pasos:

Para cada objeto que queríamos clasificar, calculamos su distancia a todos los puntos en nuestro conjunto de entrenamiento.

Luego, seleccionamos los K puntos más cercanos.

Finalmente, clasificamos el objeto en cuestión según la clase más común entre estos K vecinos.

El valor de K se seleccionó en 5 para garantizar una elección óptima que minimice el error de predicción.



Identificar orden

En nuestro caso no necesitamos la predicción de una sola caja si no que se tiene una pila de las mismas, por lo que hay que preprocesar la imagen antes de utilizar k-Means o KNN. Primero, dividimos la imagen compuesta en cuatro cuadros iguales. Luego, procesamos y clasificamos cada cuadro de forma independiente. De esta manera, pudimos asignar una clasificación para cada una de las piezas metálicas en la imagen compuesta, garantizando una identificación precisa y detallada de las piezas en nuestras imágenes.

Así podemos crear una lista con las predicciones de las cajas empezando por la que se encuentra en la parte superior y obtener el orden final

Cambiar Orden

En esta fase, el foco está en la reorganización de cajas, dado que ya se ha determinado su orden actual. Para llevar a cabo esta tarea, implementamos el lenguaje de planificación PDDL (Planning Domain Definition Language). PDDL es una extensión del lenguaje de planificación STRIPS, ambos fundamentados en la teoría de planificación basada en acciones. En PDDL, cada acción se describe a través de una "fórmula de acción", que detalla los cambios que ocurren en el estado del mundo al llevar a cabo la acción. Los cambios se definen en función de las precondiciones necesarias para que se pueda realizar la acción y los efectos en el estado del mundo cuando se ejecuta la acción.

Para abordar el problema planteado, necesitamos definir un conjunto de acciones y estados conocido como dominio. Además, es crucial formular el problema mediante las condiciones iniciales y el objetivo. Una vez que hemos planteado todo esto, pasamos a la resolución del problema utilizando un solucionador de planificación PDDL en internet denominado: Fast-Downward Web Service (lincoln.ac.uk).

En la implementación específica de ordenamiento de cajas que presentamos, hemos definido cuatro acciones principales en nuestro dominio PDDL:

- “quitar”: Esta acción permite al robot levantar una caja que está encima de otra, siempre y cuando la caja esté despejada (es decir, no haya nada encima de ella). Como resultado, la caja superior queda disponible y la caja de abajo despejada.
- “quitar-base”: Esta acción permite al robot levantar una caja que está en la base. De nuevo, esta acción solo puede llevarse a cabo si no hay nada encima de la caja. Como resultado, la caja queda disponible y la base queda despejada.
- “colocar-base”: Esta acción permite al robot poner una caja en la base. Para esto, la caja debe estar disponible y la base debe estar despejada.
- “colocar": Esta acción permite al robot poner una caja encima de otra. Para esto, la caja a colocar debe estar disponible y la caja sobre la cual se va a colocar debe estar despejada.

Estas acciones permiten al robot manipular las cajas y obtener el orden deseado.

Además, hemos definido varios predicados que describen el estado del mundo. Estos predicados representan relaciones entre los objetos en nuestro mundo (que en este caso son las cajas y la base). Los predicados incluyen:

- “encima”, que indica que una caja está sobre otra.
- “disponible", que indica que una caja está disponible.
- “nadaSobre”, que indica que no hay nada sobre una caja o la base.
- “baseCaja”, que indica que una caja está en la base.
- “posicionBase”, que indica la presencia de la base.
- “clear”, que indica que la base está despejada.
- “colocado”, que indica que una caja ha sido colocada.
- “cajaObj”, que identifica los objetos que son cajas.

Los objetos en nuestro problema PDDL son las cajas (arandelas, tuercas, clavos, tornillos) y la base (base1). Estos objetos son los que el robot puede manipular y sobre los cuales se aplican las acciones.

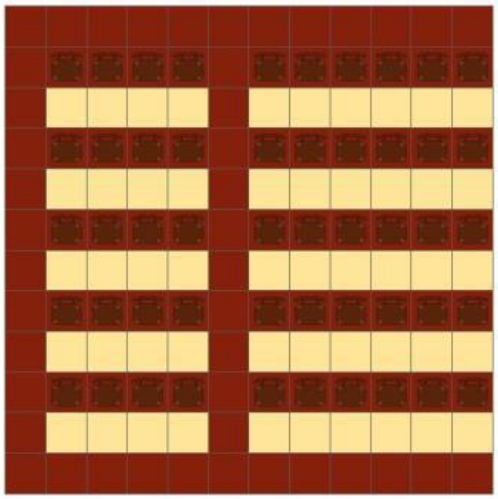
Finalmente, formulamos el problema específico a resolver en términos de un estado inicial y un estado objetivo. El estado inicial describe cómo están las cajas al principio, predicho en el paso anterior, y el estado objetivo describe cómo queremos que estén las cajas al final. Una vez que tenemos todo esto, podemos usar el solucionador de planificación para encontrar una secuencia de acciones que nos permita pasar del estado inicial al estado objetivo, resolviendo así nuestro problema.

Reubicar pila

En esta etapa del proceso, las cajas han sido debidamente organizadas y ahora es el turno de moverlas de un lugar a otro. En esta tarea, utilizamos el algoritmo de búsqueda A*, un método eficiente y efectivo para encontrar el camino más corto entre dos puntos en un almacén.

El algoritmo de búsqueda A* es un algoritmo de búsqueda que encuentra el camino más corto entre los nodos inicial y final de un gráfico ponderado. A* es un algoritmo informado, lo que significa que utiliza conocimiento sobre el problema para buscar de manera más eficiente. En particular, utiliza una función heurística para estimar el costo desde un nodo dado hasta el nodo final, y utiliza esta estimación junto con el costo de llegar a dicho nodo para guiar su búsqueda.

El mapa del almacén en donde se deben mover las cajas se muestra a continuación



La implementación del algoritmo A* en nuestro caso es como sigue:

Definimos un nodo inicial que representa el estante en donde están las cajas en una primera instancia, y un nodo final que indica el estante al que se quiere llevar. Cabe destacar que los estantes están numerados empezando por el de la izquierda arriba y hacia la derecha de manera descendente.

Para cada nodo, empezando por el inicial, calculamos sus nodos vecinos y calculamos su valor de función f, que es la suma del costo hasta ahora (g) y la estimación heurística del costo para llegar al nodo final (h). Los nodos vecinos son aquellos que se pueden alcanzar desde el nodo actual, es decir, aquellos que no son obstáculos y que no han sido visitados aún.

Elegimos el nodo con el menor valor de f de todos los nodos abiertos (es decir, no visitados aún), y lo marcamos como visitado.

Repetimos esto hasta que el nodo final es visitado o no quedan nodos abiertos.

Una vez que llegamos al nodo final, utilizamos la información del padre de cada nodo para rastrear la ruta desde el nodo final al nodo inicial, esto se hace para eliminar posibles nodos visitados que no pertenecen a la ruta más optima.

En el código, calculamos los vecinos de cada nodo considerando movimientos horizontales y verticales (sin movimientos diagonales y calculamos el costo f como la suma del costo actual (r) y la distancia Manhattan (nuestra función heurística) al nodo final. Los nodos se

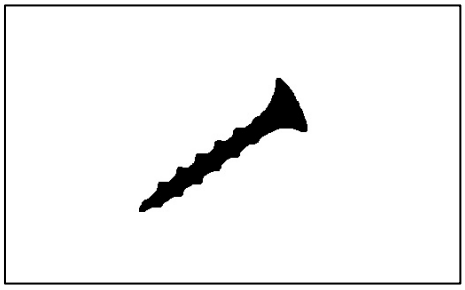
mantienen en una lista abierta que se ordena por el valor de f, asegurando que siempre estamos eligiendo el nodo más prometedor para visitar a continuación.

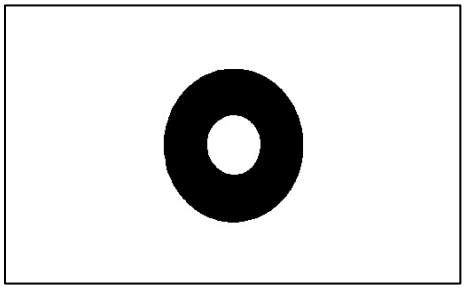
Este enfoque garantiza que el robot siempre tomará la ruta más corta para mover una caja de un lugar a otro, minimizando el tiempo y esfuerzo requerido.

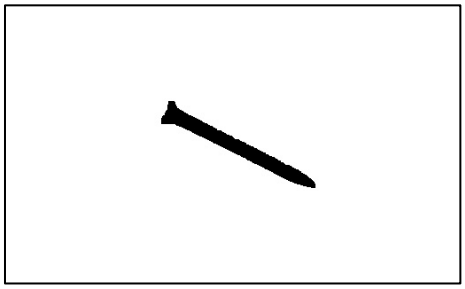
Ejemplo de Aplicación

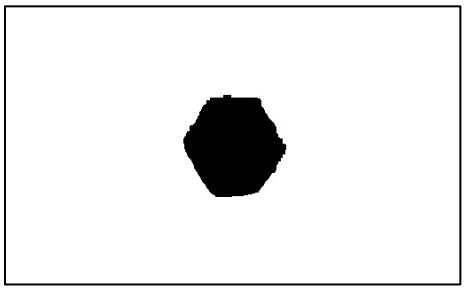
Se toma una fotografía de las cajas apiladas como se muestra a continuación:

Las imágenes completas preprocesadas para calcular los momentos de Hu son las siguientes:

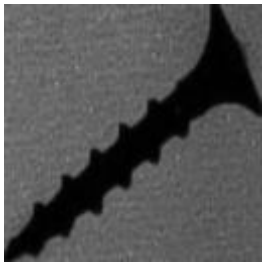


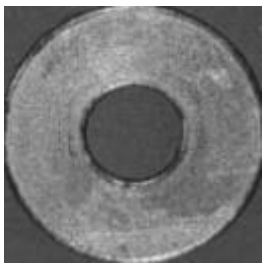


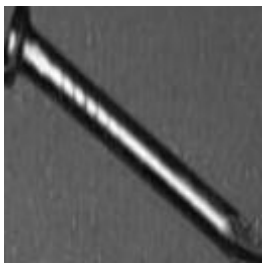


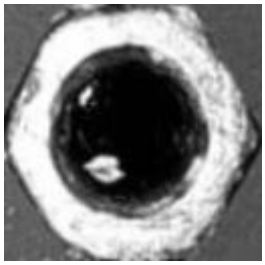


Y las imágenes preprocesadas para la caracterización por HOG son:









Luego de realizar la caracterización y ejecutar el programa obtenemos los siguientes valores:

Para el modelo K-Means:

```
Predicciones hog:
Tornillos
Arandelas
Clavos
Tuercas

Predicciones hu:
Tornillos
Arandelas
Clavos
Tuercas
```

Para el modelo KNN:

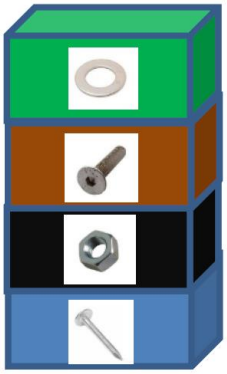
```
Predicciones hog:
Tornillos
Arandelas
Clavos
Tuercas

Predicciones hu:
Tornillos
Arandelas
Clavos
Tuercas
```

Notamos que el algoritmo predijo exitosamente objeto de las cajas y el respectivo orden para las 2 caracterizaciones y los dos métodos de clasificación.

Una vez identificado el orden, lo introducimos como estado inicial en la siguiente etapa donde se encuentra una lista de pasos para llegar a un estado final.

El estado final escogido es el que se puede visualizar en las consignas del proyecto:

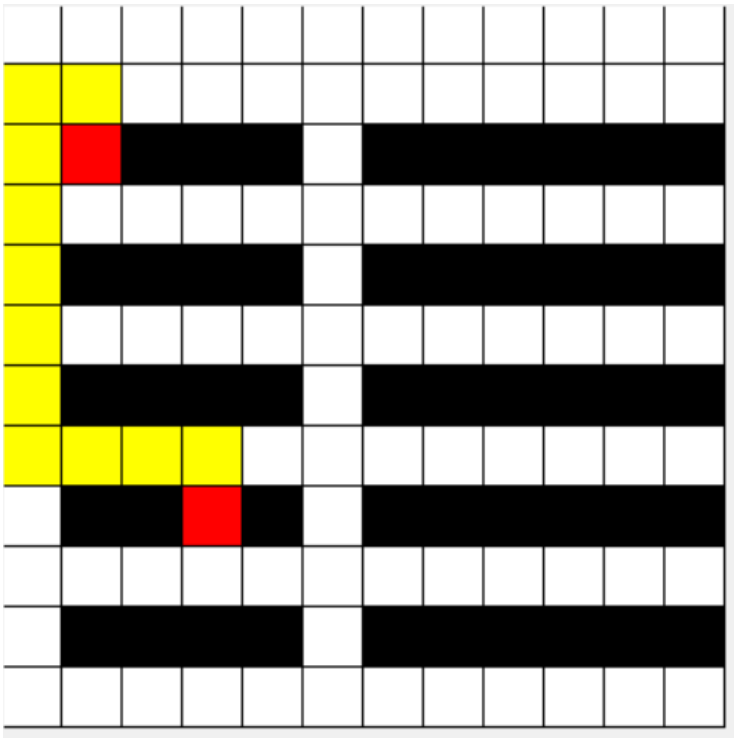


Los resultados obtenidos son son:

Plan

(quitar tornillo arandela)
(quitar arandela clavo)
(quitar clavo tuerca)
(quitar-base tuerca base1)
(colocar-base clavo base1)
(colocar tuerca clavo)
(colocar tornillo tuerca)
(colocar arandela tornillo)
; cost = 8 (unit cost)

Finalmente, para mover la pila de cajas elegimos como nodo inicial el primer estante y como nodo final el estante 33 (elegido de forma aleatoria). El resultado gráficamente se puede ver a continuación:



Implementación

MAIN.PY

```
from Preprocesamiento import Preprocesamiento
import cv2

import warnings
warnings.filterwarnings("ignore")

def main():
    preprocessor = Preprocesamiento()

    while True:
        print("\n-----------------------------------")
        print("1. Preprocesar y guardar datos.")
        print("2. Cargar datos existentes.")
        print("3. Clasificar imagen con Kmeans.")
        print("4. Clasificar imagen con KNN.")
        print("5. Clasificar una imagen compuesta con KMeans o KNN.")
        print("6. Salir.")
        print("-----------------------------------")

        opcion = int(input("Ingrese el número de la opción que desea
realizar: "))

        if opcion == 1:
            preprocessor.preprocesar_datos()
            print("Los datos han sido preprocesados y guardados con éxito.")

        elif opcion == 2:
            preprocessor.cargar_datos()
            print("Los datos han sido cargados con éxito.")

        elif opcion == 3:
            imagen_path = "D:\\FACULTAD\\MATERIAS PENDIENTES\\92 Inteligencia
Artificial\\Proyecto\\Base de datos Test\\TUERCA.jpeg"
            imagen = cv2.imread(imagen_path)
            if imagen is None:
                print("No se pudo abrir la imagen. Por favor, intenta de
nuevo.")
                continue
            preprocessor.clasificar_imagen(imagen, 'kmeans')

        elif opcion == 4:

            imagen_path = "D:\\FACULTAD\\MATERIAS PENDIENTES\\92 Inteligencia
Artificial\\Proyecto\\Base de datos Test\\TUERCA.jpeg"
```

```
            imagen = cv2.imread(imagen_path)
            if imagen is None:
                print("No se pudo abrir la imagen. Por favor, intenta de
nuevo.")
                continue
            preprocessor.clasificar_imagen(imagen, 'knn')

        elif opcion == 5:
            metodo = input("Ingrese el método a usar (kmeans/knn): ")

            imagen_path = "D:\\FACULTAD\\MATERIAS PENDIENTES\\92 Inteligencia
Artificial\\Proyecto\\Base de datos Test\\12345.jpg"
            imagen = cv2.imread(imagen_path)
            if imagen is None:
                print("No se pudo abrir la imagen. Por favor, intenta de
nuevo.")
                continue
            predicciones = preprocessor.Preprocesar_multiple(imagen, metodo)

            print("\nPredicciones hog: ")
            for pred in [predicciones[i][0] for i in range(4)]:
                print(pred)

            print("\nPredicciones hu: ")
            for pred in [predicciones[i][1] for i in range(4)]:
                print(pred)

        elif opcion == 6:
            print("Saliendo...")
            break

if __name__ == "__main__":
    main()
```

PREPROCESAR.PY

```
import os
import cv2
import numpy as np
import pickle
import matplotlib.pyplot as plt
from skimage.feature import hog
from sklearn.decomposition import PCA
from Knn import Knn
from Kmeans import KMeans
from Hu_Moments import Hu_Moments

class Preprocesamiento:
    def __init__(self):
```

```
        self.Carpeta = 'D:\FACULTAD\MATERIAS PENDIENTES\92 Inteligencia
Artificial\Proyecto\Base de datos'

        self.Categorias = ['Tornillos', 'Tuercas', 'Clavos', 'Arandelas']
        self.Etiquetas = ['Tornillos'] * 8 + ['Tuercas'] * 8 + ['Clavos'] * 8
+ ['Arandelas'] * 8

        self.Features_hog = []
        self.Features_hu = []

        self.models_kmeans = []
        self.models_knn = []

    def preprocesar_datos(self):

        self.Features_hog = []
        self.Features_hu = []

        self.models_kmeans = []
        self.models_knn = []

        kernel = np.ones((5,5),np.uint8)

        for categoria in self.Categorias:
            Subcarpeta = os.path.join(self.Carpeta, categoria)
            for NombreImagen in os.listdir(Subcarpeta):
                print("Procesando imagen: ", NombreImagen)
                img = cv2.imread(os.path.join(Subcarpeta, NombreImagen))
                img1=img.copy()
                img = img[20:-20,20:-20]
                img2=img.copy()
                img = cv2.resize(img, (512, 312))
                img3=img.copy()
                img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
                img4=img_gray.copy()
                img_blur = cv2.GaussianBlur(img_gray, (5, 5), 0)
                img5=img_blur.copy()
                img_hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
                img6=img_hsv.copy()
                lower_range = np.array([140, 50, 50])
                upper_range = np.array([160, 255, 255])

                mask = cv2.inRange(img_hsv, lower_range, upper_range)
                img7=mask.copy()
                mask = cv2.bitwise_not(mask)
                img8=mask.copy()
                img_masked = cv2.bitwise_and(img_blur, img_blur, mask=mask)
                img9=img_masked.copy()
```

```
                img_masked = cv2.morphologyEx(img_masked, cv2.MORPH_CLOSE,
kernel)

                contours, _ = cv2.findContours(img_masked, cv2.RETR_TREE,
cv2.CHAIN_APPROX_SIMPLE)

                img_new = np.ones(img_gray.shape, dtype=np.uint8)*255

                cv2.drawContours(img_new, contours, -1, (0),
thickness=cv2.FILLED)
                img10=img_new.copy()
                img_new = cv2.dilate(img_new, kernel, iterations = 1)
                img_new = cv2.erode(img_new, kernel, iterations = 1)
                img11=img_new.copy()


                M = cv2.moments(contours[0])

                try:
                    cX = int(M["m10"] / M["m00"])
                    cY = int(M["m01"] / M["m00"])
                except:
                    try:
                        M = cv2.moments(contours[1])
                        cX = int(M["m10"] / M["m00"])
                        cY = int(M["m01"] / M["m00"])
                    except:
                        M = cv2.moments(contours[2])
                        cX = int(M["m10"] / M["m00"])
                        cY = int(M["m01"] / M["m00"])

                height, width = img_new.shape[:2]
                dX = (width // 2) - cX
                dY = (height // 2) - cY

                T = np.float32([[1, 0, dX], [0, 1, dY]])
                img_new = cv2.warpAffine(img_new, T, (width, height),
borderValue = 255)
                img12=img_new.copy()
                moments = cv2.moments(img_new)
                hu_moments = Hu_Moments(moments)

                bounding_rect = cv2.boundingRect(contours[0])
                img = img[bounding_rect[1]:bounding_rect[1]+bounding_rect[3],
bounding_rect[0]:bounding_rect[0]+bounding_rect[2]]
                img = cv2.resize(img, (128, 128),
interpolation=cv2.INTER_LINEAR)
                img13=img.copy()
                img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

```
                hog_features = hog(img_gray, orientations=9,
pixels_per_cell=(8, 8), cells_per_block=(2, 2), transform_sqrt=True,
block_norm="L2")

                self.Features_hog.append(hog_features)
                self.Features_hu.append(hu_moments)

                NombreImagen = "\ " + NombreImagen + ".jpeg"
                direccion1 = "D:\FACULTAD\MATERIAS PENDIENTES\92 Inteligencia
Artificial\Proyecto\Fotos viejas" + NombreImagen
                cv2.imwrite(direccion1, img_new)

                direccion2 = "D:\FACULTAD\MATERIAS PENDIENTES\92 Inteligencia
Artificial\Proyecto\Fotos viejas2" + NombreImagen
                cv2.imwrite(direccion2, img_gray)

        lista_imagenes =
[img1,img2,img3,img4,img5,img6,img7,img8,img9,img10,img11,img12,img13]
        carpeta_destino = "D:\FACULTAD\MATERIAS PENDIENTES\92 Inteligencia
Artificial\Proyecto\Para trabajo"
        if not os.path.exists(carpeta_destino):
            os.makedirs(carpeta_destino)

        for i, img in enumerate(lista_imagenes):
            cv2.imwrite(os.path.join(carpeta_destino,
'imagen_{}.jpeg'.format(i)), img)

        self.Features_hog = np.array(self.Features_hog)
        self.Features_hu = np.array(self.Features_hu)

        self.models_kmeans = []
        self.models_knn = []

        for i, features in enumerate([self.Features_hog, self.Features_hu]):

            features = np.array(features)
            model_kmeans = KMeans()
            model_kmeans.fit(features)

            etiquetas_centroides = []

            for centroide in model_kmeans.centroides_:

                distancias = [np.linalg.norm(centroide - punto) for punto in
features]
                indices_mas_cercanos = np.argsort(distancias)[:5]
                etiquetas_mas_cercanas = [self.Etiquetas[i] for i in
indices_mas_cercanos]
```

```
                etiqueta_centroide = max(set(etiquetas_mas_cercanas),
key=etiquetas_mas_cercanas.count)
                etiquetas_centroides.append(etiqueta_centroide)
            model_kmeans.labels_ = etiquetas_centroides
            model_knn = Knn()

            model_knn.fit(list(zip(features, self.Etiquetas)))

            self.models_kmeans.append(model_kmeans)
            self.models_knn.append(model_knn)

        file_paths = ['knn_models.p','kmeans_models.p', 'Features_hog.p',
'Features_hu.p']

        for file_path in file_paths:
            if os.path.exists(file_path):
                os.remove(file_path)

        with open('knn_models.p', 'wb') as f:
            pickle.dump(self.models_knn, f)
        with open('kmeans_models.p', 'wb') as f:
            pickle.dump(self.models_kmeans, f)
        with open('Features_hog.p', 'wb') as f:
            pickle.dump(self.Features_hog, f)
        with open('Features_hu.p', 'wb') as f:
            pickle.dump(self.Features_hu, f)

    def cargar_datos(self):

        self.Features_hog = []
        self.Features_hu = []

        self.models_kmeans = []
        self.models_knn = []

        with open('Features_hog.p', 'rb') as f:
            self.Features_hog = pickle.load(f)
        with open('Features_hu.p', 'rb') as f:
            self.Features_hu = pickle.load(f)
        with open('kmeans_models.p', 'rb') as f:
            self.models_kmeans = pickle.load(f)
        with open('knn_models.p', 'rb') as f:
            self.models_knn = pickle.load(f)

    def clasificar_imagen(self, imag, method):

        hog_features, hu_moments = self.preprocesamiento_simple(imag)

        colors = ['b', 'g', 'r', 'c','m']
```

```
        categorias = ['Tornillos', 'Tuercas', 'Clavos', 'Arandelas']

        resultados = []

        if method == 'kmeans':
            for i, (features_nuevas, features_existentes, metodo) in
enumerate(zip([hog_features, hu_moments], [self.Features_hog,
self.Features_hu], ['HOG', 'Hu Moments'])):

                distancias = [np.linalg.norm(features_nuevas - centroide) for
centroide in (self.models_kmeans[i]).centroides_]

                indice_centroide_cercano = np.argmin(distancias)

                features_nuevas_nombre =
(self.models_kmeans[i]).labels_[indice_centroide_cercano]
                resultados.append(features_nuevas_nombre)

                pca = PCA(n_components=3)
                pca.fit(np.array(features_existentes))

                reduced_features = pca.transform(features_existentes)
                reduced_features_nuevas =
pca.transform(features_nuevas.reshape(1, -1))
                reduced_centroides =
pca.transform((self.models_kmeans[i]).centroides_)

                fig = plt.figure(figsize=(10, 6))
                ax = fig.add_subplot(111, projection='3d')
                for j, (centroid, color) in enumerate(zip(reduced_centroides,
colors)):
                    points =
reduced_features[(self.models_kmeans[i]).clusters == j]
                    ax.scatter(points[:, 0], points[:, 1], points[:, 2],
color=color, alpha=0.5)
                    ax.scatter(centroid[0], centroid[1], centroid[2],
color='k', marker='x', s=100)
                    ax.text(centroid[0], centroid[1], centroid[2],
(self.models_kmeans[i]).labels_[j] , fontsize=12)

                ax.scatter(reduced_features_nuevas[0, 0],
reduced_features_nuevas[0, 1], reduced_features_nuevas[0, 2],
color='magenta', marker='*', s=200, label=f"Predicción:
{features_nuevas_nombre}")
                plt.title(f"Kmeans para {metodo}")
                plt.legend()
                plt.show()

            return resultados
```

```
        elif method == 'knn':
            for i, (features_nuevas, features_existentes, metodo) in
enumerate(zip([hog_features, hu_moments],[self.Features_hog,
self.Features_hu], ['HOG', 'Hu Moments'])):

                label_asignado =
(self.models_knn[i]).predict(features_nuevas)
                resultados.append(label_asignado)

                features_existentes_np = np.array(features_existentes)

                pca = PCA(n_components=3)
                pca.fit(features_existentes_np)

                reduced_features = pca.transform(features_existentes_np)
                reduced_features_nuevas =
pca.transform(features_nuevas.reshape(1, -1))

                fig = plt.figure(figsize=(10, 6))
                ax = fig.add_subplot(111, projection='3d')

                cantidad_puntos = 8

                inicio = 0
                for color, categoria in zip(colors, categorias):
                    puntos = reduced_features[inicio: inicio +
cantidad_puntos]
                    ax.scatter(puntos[:, 0], puntos[:, 1], puntos[:, 2],
alpha=0.5, color=color, label=categoria)
                    inicio += cantidad_puntos

                for label in set(self.Etiquetas[inicio:]):
                    points = reduced_features[self.Etiquetas[inicio:] ==
label]
                    ax.scatter(points[:, 0], points[:, 1], points[:, 2],
alpha=0.5, label=label)

                ax.scatter(reduced_features_nuevas[0][0],
reduced_features_nuevas[0][1], reduced_features_nuevas[0][2],
color='magenta', marker='*', s=200, label=f"Predicción: {label_asignado}")
                plt.title(f"KNN para {metodo}")
                plt.legend()
                plt.show()

            return resultados


    def preprocesamiento_simple(self, imag):
```

```
        kernel = np.ones((5,5),np.uint8)
        imag = imag[20:-20,20:-20]

        imag = cv2.resize(imag, (512, 312))

        imag_gray = cv2.cvtColor(imag, cv2.COLOR_BGR2GRAY)

        imag_blur = cv2.GaussianBlur(imag_gray, (5, 5), 0)

        img_hsv = cv2.cvtColor(imag, cv2.COLOR_BGR2HSV)

        lower_range = np.array([140, 50, 50])
        upper_range = np.array([160, 255, 255])

        mask = cv2.inRange(img_hsv, lower_range, upper_range)
        mask = cv2.bitwise_not(mask)
        img_masked = cv2.bitwise_and(imag_blur, imag_blur, mask=mask)
        img_masked = cv2.morphologyEx(img_masked, cv2.MORPH_CLOSE, kernel)

        contours, _ = cv2.findContours(img_masked, cv2.RETR_TREE,
cv2.CHAIN_APPROX_SIMPLE)

        img_new = np.ones(imag_gray.shape, dtype=np.uint8)*255
        cv2.drawContours(img_new, contours, -1, (0), thickness=cv2.FILLED)
        img_new = cv2.dilate(img_new, kernel, iterations = 1)
        img_new = cv2.erode(img_new, kernel, iterations = 1)
        M = cv2.moments(contours[0])

        try:
            cX = int(M["m10"] / M["m00"])
        except:
            try:
                M = cv2.moments(contours[1])
                cX = int(M["m10"] / M["m00"])
            except:
                M = cv2.moments(contours[2])
                cX = int(M["m10"] / M["m00"])
        cY = int(M["m01"] / M["m00"])

        height, width = img_new.shape[:2]
        dX = (width // 2) - cX
        dY = (height // 2) - cY

        T = np.float32([[1, 0, dX], [0, 1, dY]])
        img_new = cv2.warpAffine(img_new, T, (width, height), borderValue =
255)

        moments = cv2.moments(img_new)
```

```
        hu_moments = Hu_Moments(moments)


        bounding_rect = cv2.boundingRect(contours[0])
        imag = imag[bounding_rect[1]:bounding_rect[1]+bounding_rect[3],
bounding_rect[0]:bounding_rect[0]+bounding_rect[2]]
        imag = cv2.resize(imag, (128, 128), interpolation=cv2.INTER_LINEAR)


        imag_gray = cv2.cvtColor(imag, cv2.COLOR_BGR2GRAY)
        hog_features = hog(imag_gray, orientations=9, pixels_per_cell=(8, 8),
cells_per_block=(2, 2), transform_sqrt=True, block_norm="L2")

        print(hog_features)
        print(hu_moments)

        return np.array(hog_features), np.array(hu_moments)

    def Preprocesar_multiple(self, imagen,metodo):

        altura, ancho = imagen.shape[:2]

        imagen_ver = cv2.resize(imagen, (400, 600))
        cv2.imshow(f"Foto a predecir", imagen_ver)
        cv2.waitKey(0)
        cv2.destroyAllWindows()

        imgs = [imagen[i*altura//4:(i+1)*altura//4, :] for i in range(4)]
        imgs_recortadas = []
        imgs_recortadas.append(imgs[0][10:-10,10:-10])
        imgs_recortadas.append(imgs[1][10:-10,10:-10])
        imgs_recortadas.append(imgs[2][10:-10,10:-10])
        imgs_recortadas.append(imgs[3][10:-10,10:-10])

        cv2.imwrite("D:\\FACULTAD\\MATERIAS PENDIENTES\\92 Inteligencia
Artificial\\Proyecto\\Base de datos Test\\Caja-1.jpeg", imgs_recortadas[0])
        cv2.imwrite("D:\\FACULTAD\\MATERIAS PENDIENTES\\92 Inteligencia
Artificial\\Proyecto\\Base de datos Test\\Caja-2.jpeg", imgs_recortadas[1])
        cv2.imwrite("D:\\FACULTAD\\MATERIAS PENDIENTES\\92 Inteligencia
Artificial\\Proyecto\\Base de datos Test\\Caja-3.jpeg", imgs_recortadas[2])
        cv2.imwrite("D:\\FACULTAD\\MATERIAS PENDIENTES\\92 Inteligencia
Artificial\\Proyecto\\Base de datos Test\\Caja-4.jpeg", imgs_recortadas[3])

        predicciones = []
        i=1
        for img in imgs_recortadas:

            cv2.imshow(f"Caja {i}", img)
            cv2.waitKey(0)
```

```
            cv2.destroyAllWindows()

            predicciones.append(self.clasificar_imagen(img, metodo))
            i+=1
        return predicciones
```

KMEANS.PY

```
import numpy as np

class KMeans:
    def __init__(self, n_clusters=4):

        self.n_clusters = n_clusters
        self.centroides_ = None
        self.labels_ = None
        self.clusters = None

    def fit(self, X):

        self.centroides_ = self._kmeans_plusplus(X)
        prev_centroids = np.zeros_like(self.centroides_)

        while np.linalg.norm(self.centroides_ - prev_centroids) > 1e-4:
            self.clusters = np.argmin([np.linalg.norm(X - c, axis=1) for c in
self.centroides_], axis=0)
            prev_centroids = self.centroides_.copy()
            self.centroides_ = np.array([X[self.clusters == k].mean(axis=0)
for k in range(self.n_clusters)])
        return self

    def _kmeans_plusplus(self, X):
        n, _ = X.shape
        centroides = np.zeros((self.n_clusters, X.shape[1]))
        initial_idx = np.random.choice(n)
        centroides[0] = X[initial_idx]
        for i in range(1, self.n_clusters):
            dist_sq = np.min([np.linalg.norm(X - c, axis=1)**2 for c in
centroides[:i]], axis=0)
            probs = dist_sq / np.sum(dist_sq)
            cumulative_probs = np.cumsum(probs)
            idx = np.where(cumulative_probs >= np.random.rand())[0][0]
            centroides[i] = X[idx]
        return centroides
```

KNN.PY

```
import numpy as np
from collections import Counter
```

```
class Knn:
    def __init__(self):
        self.k = 5

    def fit(self, data):
        self.data = data

    def euclidean_distance(self, x1, x2):
        return np.sqrt(np.sum((x1 - x2)**2))

    def predict(self, img):
        y_pred = self._predict(img)
        return y_pred

    def _predict(self, img):
        distances = [self.euclidean_distance(img, data[0]) for data in
self.data]
        k_indices = np.argsort(distances)[:self.k]
        k_nearest_labels = [self.data[i][1] for i in k_indices]
        most_common = Counter(k_nearest_labels).most_common(1)
        return most_common[0][0]
```

HU_MOMENTS.PY

```
def Hu_Moments(moments):
    # Extraer los momentos centrales desde el diccionario de momentos
    m00 = moments['m00']
    m10 = moments['m10']
    m01 = moments['m01']
    m20 = moments['m20']
    m02 = moments['m02']
    m11 = moments['m11']
    m30 = moments['m30']
    m21 = moments['m21']
    m12 = moments['m12']
    m03 = moments['m03']

    # Calcular el centroide
    xc = m10 / m00
    yc = m01 / m00

    # Calcular los momentos centrales normalizados
    n20 = m20 / m00**2
    n02 = m02 / m00**2
    n11 = m11 / m00**2
    n30 = m30 / m00**2.5
    n12 = m12 / m00**2.5
    n21 = m21 / m00**2.5
```

```
    n03 = m03 / m00**2.5

    phi1 = n20 + n02
    phi2 = (n20 - n02)**2 + 4*n11**2
    phi3 = (n30 - 3*n12)**2 + (3*n21 - n03)**2
    phi4 = (n30 + n12)**2 + (n21 + n03)**2
    phi5 = (n30 - 3*n12) * (n30 + n12) * ((n30 + n12)**2 - 3*(n21 + n03)**2)
+ (3*n21 - n03) * (n21 + n03) * (3*(n30 + n12)**2 - (n21 + n03)**2)
    phi6 = (n20 - n02) * ((n30 + n12)**2 - (n21 + n03)**2) + 4*n11 * (n30 +
n12) * (n21 + n03)
    phi7 = (3*n21 - n03) * (n30 + n12) * ((n30 + n12)**2 - 3*(n21 + n03)**2)
- (n30 - 3*n12) * (n21 + n03) * (3*(n30 + n12)**2 - (n21 + n03)**2)

    Hu = [phi1, phi2, phi3, phi4, phi5, phi6, phi7]
    return Hu
```

AESTRELLA.PY

```
import matplotlib.pyplot as plt
from Grafico import Grafico
import copy
from math import dist

class Aestrella():
    def __init__(self,filas,columnas):

        self.Cantfilas = filas
        self.Cantcolumnas = columnas
        self.Obstaculos= self.CalcObstaculos()

    def calcularcamino(self,inicio,final):

        nodoinicial = self.Obstaculos[inicio-1]
        nodofinal = self.Obstaculos[final-1]
        nodofinalp = [nodofinal[0]-1,nodofinal[1]]
        nodoinicial = [nodoinicial[0]-1,nodoinicial[1]]
        nodosabiertos = []
        nodosvisitados = []
        nodoactual = nodoinicial
        nodosvisitados.append(((nodoinicial[0]),(nodoinicial[1]),0))

        if nodoactual == nodofinalp:
            return 0, [nodoinicial, nodofinalp], nodofinalp
        r=0
        while True:

            if nodoactual == nodofinalp:
                break
            r+=1
            iactual = nodoactual[0]
```

```
            jactual = nodoactual[1]
            for i in [-1,0, 1]:
                for j in [-1, 0, 1]:
                    inuevo = iactual + i
                    jnuevo = jactual + j
                    if ((inuevo, jnuevo) not in [(sublst[0], sublst[1]) for
sublst in nodosabiertos]) and ((inuevo, jnuevo) not in self.Obstaculos) and
((inuevo, jnuevo) not in [(sublst[0], sublst[1]) for sublst in
nodosvisitados]) and (inuevo >= 0) and (jnuevo >= 0) and (inuevo <= 11) and
(jnuevo <= 11) and abs(i) != abs(j):
                        f = self.heuristica(nodofinalp[0], nodofinalp[1],
inuevo, jnuevo) + r
                        nodosabiertos.append((inuevo, jnuevo,f , r, (iactual,
jactual)))

            nodosabiertos = sorted(nodosabiertos, key=lambda x: x[2])
            nodoactual[0],nodoactual[1], *_ = nodosabiertos[0]

nodosvisitados.append(((nodosabiertos[0][0]),(nodosabiertos[0][1]),(nodosabie
rtos[0][3]),(nodosabiertos[0][4])))
            nodosabiertos.pop(0)
        listafinal=[]
        while True:

            for i in range(len(nodosvisitados)-1):
                if len(nodosvisitados)==2:
                    listafinal.insert(0,(nodosvisitados[-
1][0],nodosvisitados[-1][1]))

listafinal.insert(0,(nodosvisitados[0][0],nodosvisitados[0][1]))
                    return len(listafinal),listafinal,nodofinalp
                while nodosvisitados[-i-1][3] != (nodosvisitados[-i-
2][0],nodosvisitados[-i-2][1]):
                    nodosvisitados.pop(-i-2)
                listafinal.insert(0,(nodosvisitados[-i-1][0],nodosvisitados[-
i-1][1]))
                if i ==len(nodosvisitados)-2:

listafinal.insert(0,(nodosvisitados[0][0],nodosvisitados[0][1]))
                    return len(listafinal),listafinal,nodofinalp

    def dibujar(self,nodosvisitados,nodofinalp):

Grafico1=Grafico(self.Cantfilas*6,self.Cantcolumnas*4,self.Obstaculos)
        Grafico1.dibujar_grafico(nodosvisitados,nodofinalp)

    def GraficarCamino (self,lista,lista2=0):

        nodosvisitados = []
```

```
        finales = []
        for i in range(len(lista)-1):

nodosvisitados.append((self.calcularcamino(lista[i],lista[i+1]))[1])
        Grafico2=Grafico(12,12,self.Obstaculos)
        for x in lista:
            finales.append(self.Obstaculos[x-1])
        if lista2==0:
            Grafico2.dibujar_grafico(nodosvisitados,finales)
        else:
            total_elements = 0
            for sublist in  nodosvisitados:
                total_elements += len(sublist)
            return total_elements

    def heuristica (self,x,y,j,k):
        return abs(x - j) + abs(y - k)

    def CalcObstaculos(self):

        Obstaculos = []
        veccolumnas = []
        vecfilas = [2,4,6,8,10]
        veccolumnas = [1,2,3,4,6,7,8,9,10,11]
        for j in vecfilas:
            for i in veccolumnas:
                Obstaculos.append((j,i))
        return Obstaculos

AEstrella1 = Aestrella(5,2)
inicio=1
final=33
AEstrella1.calcularcamino(inicio,final)
AEstrella1.GraficarCamino([inicio,final])

GRAFICAR.PY
import tkinter as tk

class Grafico:
    def __init__(self, filas, columnas, obstaculos):
        self.filas = filas+1
        self.columnas = columnas+1
        self.obstaculos = obstaculos

    def dibujar_grafico(self,camino,final):
        self.camino = camino
        self.final = final
        print(self.camino)
```

```
        matriz = [[0 for j in range(self.columnas-1)] for i in
range(self.filas-1)]

        for i, j in self.obstaculos:
            matriz[i][j] = 2
        r=4
        for sublist in self.camino:
            for subsublist in sublist:
                i, j = subsublist[0], subsublist[1]
                matriz[i][j] = r
            r+=1
        for i, j in self.final:
            matriz[i][j] = 3

        for i in range(self.filas-1):
            for j in range(self.columnas-1):
                if matriz[i][j] == 0:
                    matriz[i][j] = 1
        ventana = tk.Tk()
        ventana.title("Grafico")

        lienzo = tk.Canvas(ventana)

        for i in range(self.filas-1):
            for j in range(self.columnas-1):
                x1 = j * 30
                y1 = i * 30
                x2 = x1 + 30
                y2 = y1 + 30
                if matriz[i][j] == 1:
                    color = "white"
                if matriz[i][j] == 2:
                    color = "black"
                elif matriz[i][j] == 3:
                    color = "red"
                elif matriz[i][j] == 4:
                    color = "yellow"
                lienzo.create_rectangle(x1, y1, x2, y2, fill=color)

        lienzo.config(width=self.columnas * 30, height=self.filas * 30)

        lienzo.pack()
        ventana.mainloop()
```

Conclusión

La realización de este proyecto en el curso de Inteligencia Artificial I ha sido una experiencia interesante y enriquecedora. Al desarrollar y comparar los algoritmos Kmeans y Knn para clasificar objetos, he profundizado en las técnicas de aprendizaje automático y su aplicación en problemas reales. También he aprendido a utilizar el lenguaje STRIPS para planificar la reorganización de cajas, lo que me ha dado una visión práctica de la planificación automatizada y cómo resolver problemas complejos. Además, al aplicar el algoritmo A* para la navegación en un laberinto, he mejorado mis habilidades para trabajar con algoritmos de búsqueda en entornos difíciles.

Lo más interesante de este trabajo ha sido la combinación de diferentes áreas de la inteligencia artificial, como la visión por computadora, el aprendizaje automático y la planificación. Esta experiencia me ha mostrado cómo estas áreas se complementan y trabajan juntas para resolver problemas de manera más completa.

Finalmente, sobre este trabajo en particular se puede decir que el funcionamiento de Kmeans ha sido exitoso en el 100% de las oportunidades si se utilizan centroides iniciales conocidos. Si se utilizan centroides aleatorios, este porcentaje se reduce a casi el 70% ya que depende mucho de la elección de los puntos iniciales y muchas veces no converge al máximo global. Por otro lado, el algoritmo de KNN funciona de manera correcta en el 100% de las veces si las imágenes son tomadas de forma correcta. Teniendo estas consideraciones puedo decir que el algoritmo KNN es la mejor opción para esta tarea.

El algoritmo A estrella, con el mapa de almacén dado, funciona correctamente sea cual sea los puntos iniciales y finales escogidos.

El algoritmo en lenguaje PDDL funciona correctamente salvo cuando el estado inicial y final poseen la misma base o la misma base y la misma caja en la segunda posición. Si bien el conjunto de acciones que devuelve el programa lleva a una solución, esta puede ser considerada redundante en estos casos.

Bibliografías y referencias

- Dra. Ing. Selva Soledad Rivera. “Apuntes de cátedra Inteligencia Artificial I”
- Dr. Ing. Martin Marchetta. “Apuntes de cátedra Inteligencia Artificial II”
- Rusell – Norving “Inteligencia Artificial – Un Enfoque Moderno”