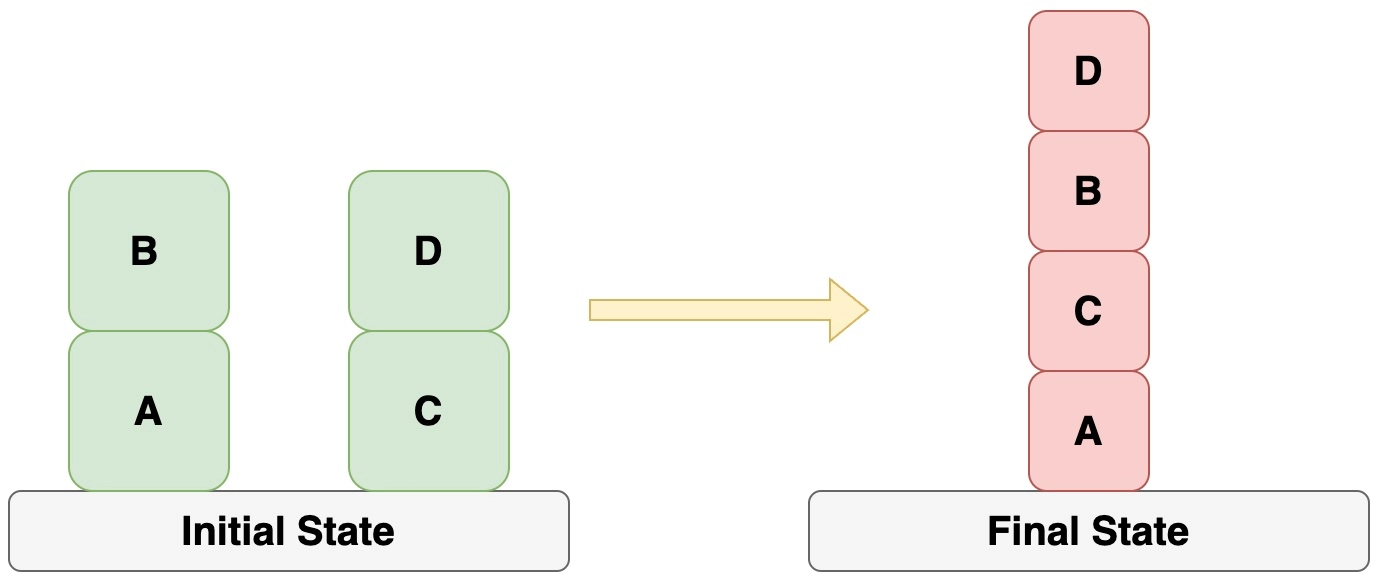
Planificación Strips



Adrián Yared Armas de la Nuez

**Contenido**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

[**1. Enunciado 2**](#_2c35568x6wg8)

[**1.1 Objetivo 2**](#_6quac0x38hdc)

[**1.2 Problema 2**](#_mqeyq2araxpk)

[**1.3 Objetivo 2**](#_uy6nbfo78pff)

[**1.4 Restricciones 2**](#_xlk5qc4r2n7i)

[**2. Tareas 2**](#_9rapxqxoggo1)

[**3. Resolución de la actividad 2**](#_8rrqphiybqku)

[**3.1 Enunciado 2**](#_rjpj17ru25h3)

[**3.1.1 Código 3**](#_3bbpit9wiqr1)

[**3.1.2 Explicación del código 3**](#_d4v8yn9ha6gb)

[**3.2 Enunciado 3**](#_z8vhicsbunb0)

[**3.2.1 Código 3**](#_5tvb09ccvplh)

[**3.2.2 Explicación del código 4**](#_4rrzg0fwuf39)

[**3.3 Enunciado 5**](#_hch7ld7mldgl)

[**3.3.1 Código 5**](#_lflodsni1h79)

[**3.3.2 Explicación del código 5**](#_9ggbo6wx574g)

[**3.4 Enunciado 5**](#_nszkshmqrv1s)

[**3.4.1 Código 5**](#_176e01cbq8yd)

[**3.4.2 Explicación del código 6**](#_vlbdua5m5tit)

[**3.5 Enunciado 6**](#_kjq7cjfefewb)

[**3.5.1 Código 6**](#_fck5pw8xd84l)

[**3.5.2 Explicación del código 7**](#_8go1g0271kmz)

[**3.6 Interpretación del resultado 7**](#_nd6fzqa5fqtv)

[**4. Código completo 7**](#_hc9t5fpactj9)

[**5 Github y Colab 12**](#_lmdngf8dlin3)

## **1. Enunciado**

### **1.1 Objetivo**

El objetivo de esta tarea es aplicar los conceptos de planificación en el contexto de la inteligencia artificial, específicamente utilizando STRIPS (Stanford Research Institute Problem Solver), para resolver un problema de manipulación de bloques en un mundo simulado. La implementación se realizará en Python.

### **1.2 Problema**

Considera un mundo donde hay cuatro bloques identificados como A, B, C, y D. Tienes tres lugares posibles donde puedes poner estos bloques, identificados como 1, 2, y 3. Inicialmente, todos los bloques están apilados en el lugar 1 en el siguiente orden: B encima de A, A encima de C, y C encima de D.

### **1.3 Objetivo**

El objetivo es llegar a un estado donde los bloques estén apilados en el lugar 3 en el orden, A encima de B, B encima de C, y C encima de D.

### **1.4 Restricciones**

Solo puedes mover un bloque a la vez.

## **2. Tareas**

1. (2 Puntos) Define el estado inicial y el estado objetivo del problema (podrá especificarse otra).

2. (2 Puntos) Implementa las acciones posibles que se pueden realizar, teniendo en cuenta las restricciones.

3. (2 Puntos) Implementa la función de transición que actualiza el estado del mundo después de realizar una acción.

4. (2 Puntos) Utiliza STRIPS para planificar una secuencia de acciones que conduzca del estado inicial al estado objetivo.

5. (2 Puntos) Implementa las funciones necesarias para ejecutar y visualizar el plan obtenido.

## **3. Resolución de la actividad**

### **3.1 Enunciado**

Define el estado inicial y el estado objetivo del problema (podrá especificarse otra).

#### **3.1.1 Código**

self.vectorInicial = [1, 2, 1, 3] # Initial state of blocks A, B, C, D

self.vectorObjetivo = [3, 3, 2, 1] # Goal state of blocks A,B, C, D

self.inicial = int(self.mascaraInicio(), 2) # Initial state in binary

self.meta = int(self.mascaraFin(), 2) # Goal state in binary

#### **3.1.2 Explicación del código**

Las líneas self.vectorInicial y self.vectorObjetivo definen las posiciones de los bloques en el estado inicial y objetivo del problema, respectivamente, como listas de números. En self.vectorInicial = [1, 2, 1, 3], los bloques A, B, C y D están en las posiciones 1, 2, 1 y 3, mientras que self.vectorObjetivo = [3, 3, 2, 1] define la meta, donde los bloques A, B, C y D deben estar en las posiciones 3, 3, 2 y 1. Estas listas se convierten en cadenas binarias mediante las funciones mascaraInicio() y mascaraFin(), y luego esas cadenas se transforman en números enteros con self.inicial y self.meta. Estos números binarios enteros son utilizados para comparar y realizar las operaciones en la búsqueda del plan.

### **3.2 Enunciado**

Implementa las acciones posibles que se pueden realizar, teniendo en cuenta las restricciones.

#### **3.2.1 Código**

def definir\_operaciones(self):

"""Defines the preconditions, effects, and additions for all possible operations."""

self.PC = [] # List of preconditions

self.E = [] # List of negative effects

self.A = [] # List of positive effects

self.operaciones = [] # List to describe operations in text

bloques = ['A', 'B', 'C', 'D'] # Blocks in the system

posiciones = [1, 2, 3] # Possible positions

# Operations to move each block to each position

for bloque\_idx, bloque in enumerate(bloques):

for origen in posiciones:

for destino in posiciones:

if origen != destino: # Avoid unnecessary movements to the same place

pc\_bin = ['0'] \* 13

e\_bin = ['1'] \* 13

a\_bin = ['0'] \* 13

# Preconditions: the block is in the origin position

pc\_bin[3 \* bloque\_idx + (origen - 1)] = '1'

# Negative effects: the block is no longer in the origin position

e\_bin[3 \* bloque\_idx + (origen - 1)] = '0'

# Positive effects: the block is now in the destination position

a\_bin[3 \* bloque\_idx + (destino - 1)] = '1'

# Convert binary lists to integers

self.PC.append(int(''.join(pc\_bin), 2))

self.E.append(int(''.join(e\_bin), 2))

self.A.append(int(''.join(a\_bin), 2))

# Save the operation in text

self.operaciones.append(f"Move {bloque} from {origen} to {destino}")

print(f"Operations defined: {len(self.PC)}")

#### **3.2.2 Explicación del código**

La función definir\_operaciones() crea todas las acciones posibles en el problema, que son los movimientos de bloques de una posición a otra. Para cada bloque, itera sobre las posiciones de origen y destino, y si son diferentes, define las precondiciones (el bloque debe estar en la posición de origen), los efectos negativos (el bloque ya no estará en la posición de origen) y los efectos positivos (el bloque se moverá a la posición de destino). Luego, convierte estos efectos en valores binarios y los guarda como enteros en las listas self.PC, self.E y self.A. Finalmente, también almacena la descripción textual de cada operación en self.operaciones.

### **3.3 Enunciado**

Implementa la función de transición que actualiza el estado del mundo después de realizar una acción.

#### **3.3.1 Código**

def aplicar\_operacion(self, estado, op):

"""Applies an operation to the state."""

if estado & self.PC[op] == self.PC[op]: # Check precondition

estado = (estado & self.E[op]) | self.A[op] # Apply effect

return estado

#### **3.3.2 Explicación del código**

La función aplicar\_operacion() actualiza el estado del mundo después de realizar una acción. Primero verifica si el estado actual cumple con las precondiciones de la operación utilizando una comparación binaria. Si la precondición es satisfecha, aplica los efectos negativos (eliminando el bloque de su posición original) y los efectos positivos (colocando el bloque en su nueva posición) mediante operaciones bit a bit. El estado resultante se calcula con el operador & para los efectos negativos y | para los positivos. Finalmente, la función devuelve el nuevo estado actualizado tras la acción.

### **3.4 Enunciado**

Utiliza STRIPS para planificar una secuencia de acciones que conduzca del estado inicial al estado objetivo.

#### **3.4.1 Código**

def buscar\_plan(self):

"""Searches for a plan using binary depth-first search."""

visitados = set()

plan = []

estados = [] # To store the intermediate states

def dfs(estado):

if estado == self.meta:

return True

visitados.add(estado)

for op in range(len(self.PC)):

nuevo\_estado = self.aplicar\_operacion(estado, op)

if nuevo\_estado not in visitados:

plan.append(op)

estados.append(nuevo\_estado) # Save the state after applying the operation

if dfs(nuevo\_estado):

return True

plan.pop()

estados.pop()

return False

if dfs(self.inicial):

return plan, estados

else:

return None, None

#### **3.4.2 Explicación del código**

El método buscar\_plan es el encargado de utilizar STRIPS para planificar una secuencia de acciones que conduzca desde el estado inicial (inicial) al estado objetivo (meta). Este proceso se implementa a través de una búsqueda en profundidad (DFS), donde se exploran las diferentes combinaciones de operaciones disponibles y se aplican para generar nuevos estados hasta encontrar una secuencia que cumpla con el objetivo.

### **3.5 Enunciado**

Implementa las funciones necesarias para ejecutar y visualizar el plan obtenido.

#### **3.5.1 Código**

# Generar el plan

plan\_binario = mundo\_binario.buscar\_plan()

# Si se encontró un plan, mostrar los últimos movimientos y visualizar el grafo

if plan\_binario:

print("Plan encontrado:")

for paso in plan\_binario:

print(mundo\_binario.operaciones[paso])

# Mostrar los últimos movimientos de cada bloque

mundo\_binario.mostrar\_ultimos\_movimientos(plan\_binario)

# Visualizar el grafo de transiciones de estados

mundo\_binario.visualizar\_grafo(plan\_binario)

else:

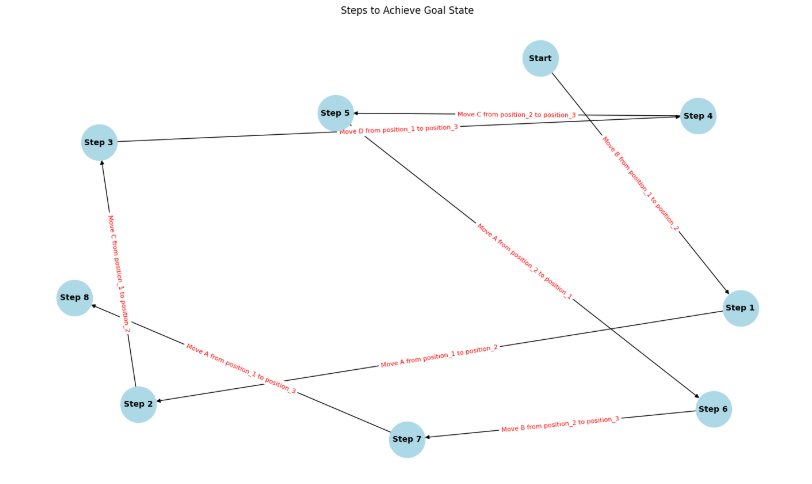
print("No se encontró un plan.")

#### **3.5.2 Explicación del código**

muestra el contenido del plan encontrado mediante la impresión de cada acción (movimiento de bloques) en el plan con la sección for paso in plan\_binario. Luego, la función mostrar\_ultimos\_movimientos(plan\_binario) imprime la última posición alcanzada por cada bloque al final del plan. Además, la función visualizar\_grafo(plan\_binario) utiliza networkx para generar y mostrar un grafo que representa las transiciones entre los estados durante la ejecución del plan. Estas funciones permiten tanto visualizar el flujo de acciones como los cambios de estado en un grafo interactivo.

### **3.6 Interpretación del resultado**

Si encuentra plan muestra lo siguiente:  
Plan found:   
Step 1: Move A from 1 to 2 (State: 0101000000000)   
Step 2: Move B from 1 to 3 (State: 0100010000000)   
Step 3: Move C from 3 to 2 (State: 0100100000000)



## **4. Código completo**

import networkx as nx

import matplotlib.pyplot as plt

class STRIPSWorldBinario:

def \_\_init\_\_(self):

"""Initializes the STRIPS world in binary format."""

self.vectorInicial = [1, 2, 1, 3] # Initial state of blocks A, B, C, D

self.vectorObjetivo = [3, 3, 2, 1] # Goal state of blocks A, B, C, D

self.PC = [] # Preconditions in binary format

self.E = [] # Negative effects in binary format

self.A = [] # Positive effects in binary format

self.operaciones = [] # List to describe operations in text

self.inicial = int(self.mascaraInicio(), 2) # Initial state in binary

self.meta = int(self.mascaraFin(), 2) # Goal state in binary

self.definir\_operaciones()

def mascaraInicio(self):

"""Generates the initial binary mask."""

mascara = ''

for pos in self.vectorInicial:

if pos == 1:

mascara += "100" # Block in position 1

elif pos == 2:

mascara += "010" # Block in position 2

elif pos == 3:

mascara += "001" # Block in position 3

return mascara + "1010" # Bits for free (C) and free (P)

def mascaraFin(self):

"""Generates the binary mask of the goal state."""

mascara = ''

for pos in self.vectorObjetivo:

if pos == 1:

mascara += "100" # Block in position 1

elif pos == 2:

mascara += "010" # Block in position 2

elif pos == 3:

mascara += "001" # Block in position 3

return mascara + "1010" # Bits for free (C) and free (P)

def definir\_operaciones(self):

"""Defines the preconditions, effects, and additions for all possible operations."""

self.PC = [] # List of preconditions

self.E = [] # List of negative effects

self.A = [] # List of positive effects

self.operaciones = [] # List to describe operations in text

bloques = ['A', 'B', 'C', 'D'] # Blocks in the system

posiciones = [1, 2, 3] # Possible positions

# Operations to move each block to each position

for bloque\_idx, bloque in enumerate(bloques):

for origen in posiciones:

for destino in posiciones:

if origen != destino: # Avoid unnecessary movements to the same place

pc\_bin = ['0'] \* 13

e\_bin = ['1'] \* 13

a\_bin = ['0'] \* 13

# Preconditions: the block is in the origin position

pc\_bin[3 \* bloque\_idx + (origen - 1)] = '1'

# Negative effects: the block is no longer in the origin position

e\_bin[3 \* bloque\_idx + (origen - 1)] = '0'

# Positive effects: the block is now in the destination position

a\_bin[3 \* bloque\_idx + (destino - 1)] = '1'

# Convert binary lists to integers

self.PC.append(int(''.join(pc\_bin), 2))

self.E.append(int(''.join(e\_bin), 2))

self.A.append(int(''.join(a\_bin), 2))

# Save the operation in text

self.operaciones.append(f"Move {bloque} from {origen} to {destino}")

print(f"Operations defined: {len(self.PC)}")

def aplicar\_operacion(self, estado, op):

"""Applies an operation to the state."""

if estado & self.PC[op] == self.PC[op]: # Check precondition

estado = (estado & self.E[op]) | self.A[op] # Apply effect

return estado

def buscar\_plan(self):

"""Searches for a plan using binary depth-first search."""

visitados = set()

plan = []

def dfs(estado):

if estado == self.meta:

return True

visitados.add(estado)

for op in range(len(self.PC)):

nuevo\_estado = self.aplicar\_operacion(estado, op)

if nuevo\_estado not in visitados:

plan.append(op)

if dfs(nuevo\_estado):

return True

plan.pop()

return False

if dfs(self.inicial):

return plan

else:

return None

def mostrar\_ultimos\_movimientos(self, plan):

"""Muestra el último movimiento de cada bloque al final del plan."""

# Inicializamos las posiciones de los bloques

posiciones = {0: '1', 1: '2', 2: '3'} # Posiciones posibles: 1, 2, 3

bloques = ['A', 'B', 'C', 'D'] # Bloques A, B, C, D

# Un diccionario para guardar la última posición de cada bloque

ultimos\_movimientos = {bloque: None for bloque in bloques}

estado\_actual = self.inicial

# Iteramos sobre el plan para actualizar las posiciones de los bloques

for paso in plan:

nuevo\_estado = self.aplicar\_operacion(estado\_actual, paso)

# Actualizamos las posiciones de los bloques

for bloque\_idx, bloque in enumerate(bloques):

for i in range(3): # Para cada bloque, comprobamos sus posiciones

if (nuevo\_estado >> (3 \* bloque\_idx + i)) & 1: # Si el bloque está en la posición i+1

ultimos\_movimientos[bloque] = posiciones[i + 1]

estado\_actual = nuevo\_estado

# Mostrar los últimos movimientos de cada bloque

print("Últimos movimientos de cada bloque:")

for bloque, posicion in ultimos\_movimientos.items():

print(f"{bloque}: Última posición - {posicion}")

def visualizar\_grafo(self, plan):

"""Visualiza el grafo de estados y acciones."""

G = nx.DiGraph() # Grafo dirigido

estado\_actual = self.inicial

G.add\_node(estado\_actual, label="Inicio")

# Agregamos los nodos y las transiciones

for paso in plan:

nuevo\_estado = self.aplicar\_operacion(estado\_actual, paso)

descripcion\_accion = self.operaciones[paso]

G.add\_node(nuevo\_estado, label=descripcion\_accion)

G.add\_edge(estado\_actual, nuevo\_estado, label=descripcion\_accion)

estado\_actual = nuevo\_estado

# Dibujar el grafo

pos = nx.spring\_layout(G) # Posicionamiento automático de los nodos

labels = nx.get\_edge\_attributes(G, 'label')

node\_labels = nx.get\_node\_attributes(G, 'label')

nx.draw(G, pos, with\_labels=True, node\_size=2000, node\_color="lightblue", font\_size=8, font\_weight="bold")

nx.draw\_networkx\_edge\_labels(G, pos, edge\_labels=labels)

nx.draw\_networkx\_labels(G, pos, labels=node\_labels, font\_size=10)

plt.show()

# Crear el mundo STRIPS

mundo\_binario = STRIPSWorldBinario()

# Generar el plan

plan\_binario = mundo\_binario.buscar\_plan()

# Si se encontró un plan, mostrar los últimos movimientos y visualizar el grafo

if plan\_binario:

print("Plan encontrado:")

for paso in plan\_binario:

print(mundo\_binario.operaciones[paso])

# Mostrar los últimos movimientos de cada bloque

mundo\_binario.mostrar\_ultimos\_movimientos(plan\_binario)

# Visualizar el grafo de transiciones de estados

mundo\_binario.visualizar\_grafo(plan\_binario)

else:

print("No se encontró un plan.")

## **5 Github y Colab**

[](https://colab.research.google.com/drive/1ifaPcBHD8sar2rjhVuwlRibolUrm1giH?usp=sharing)