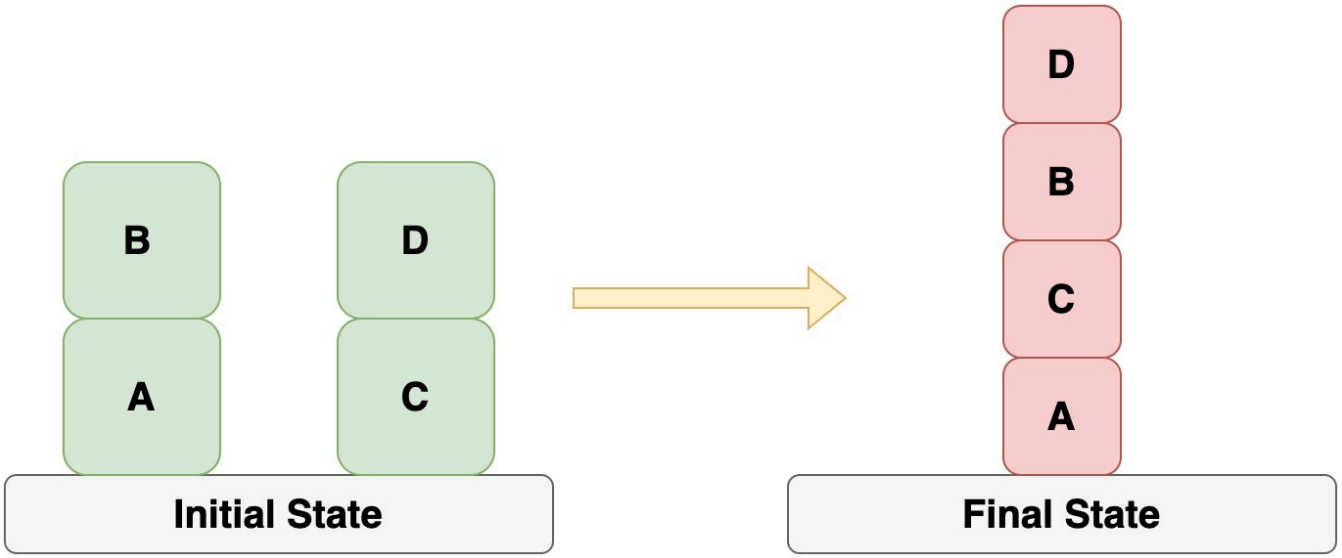
Planificación Strips



Adrián Yared Armas de la Nuez

Contenido

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
```

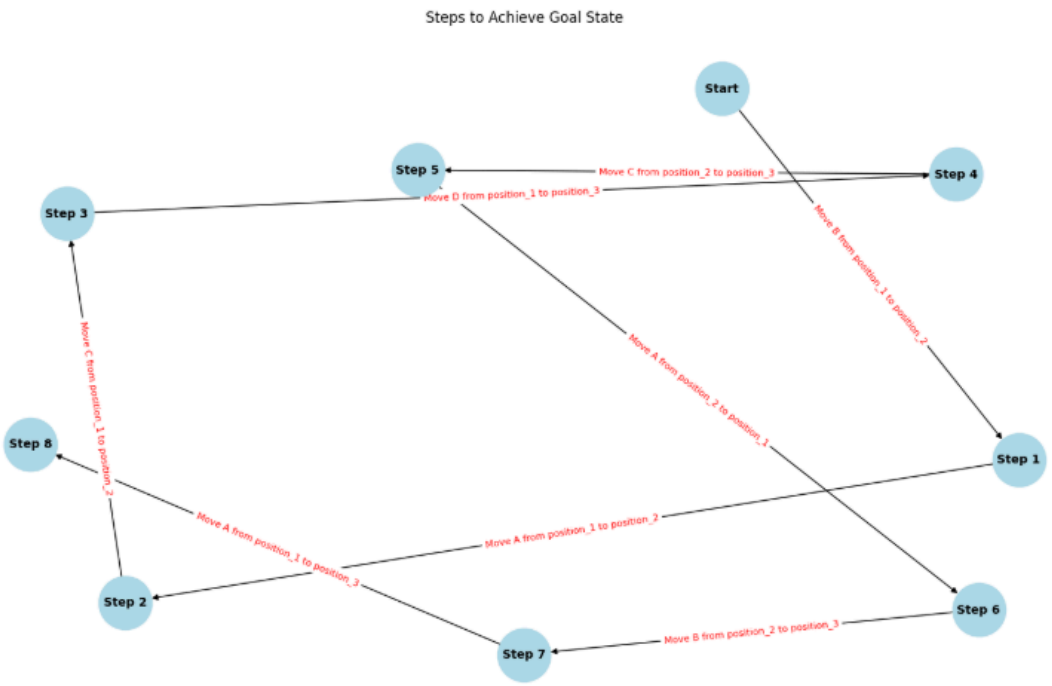
```
    mundo_binario.visualizar_grafo(plan_binario)
else:
    print("No se encontró un plan.")
```

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