Ejercicio: Identifique los componentes léxico, estructural, operativo y semántico. Adicionalmente, represente, como grafo todo el espacio de estados (identificando especialmente el estado inicial y el estado final y mostrando los cambios de un estado a otro)

en el problema de Guarini cuyo enunciado es el siguiente:

"*Uno de los problemas más antiguos en Europa relacionados con el tablero de ajedrez fue planteado en 1512. Sobre un tablero con nueve cuadros, los dos caballos blancos tienen que cambiar de sitio con dos caballos negros (usando únicamente el movimiento del caballo). ¿Cuál es el menor número de movimientos que se necesitan para hacerlo?* ". Fuente "Snape Ch. & Scott H., Desafíos Matemáticos, Ed, Limusa".

Componentes léxicos

Tablet (Matriz)

▪ ▪ ▪
▪ ▪ ▪
▪ ▪ ▪

Caballo negro

♞

Caballo blanco

♘

Componente estructural

Matriz cuadrada de 3x3

▪ ▪ ▪
▪ ▪ ▪
▪ ▪ ▪

Dos caballos blancos

♘ ▪ ♘

Dos caballos negros

♞ ▪ ♞

Movimientos posibles en la matriz

▪ ▪ ▪
▪ ▪ ▪
▪ ▪ ▪

Componente estructural

Estado inicial
Caballos negros ubicados en las ezquinas superiores
Caballos blancos ubicados en esquinas inferiores

♞ · ♞
· · ·
♘ · ♘

Estado final
Caballos negros ubicados en las ezquinas inferiores
Caballos blancos ubicados en esquinas superiores

♘ · ♘
· · ·
♞ · ♞

Desarrollo de la solución

La solución para el problema se describe en los siguientes pasos bien definidos a continuación, el programa fué desarrollado en python acorde con los enunciados del ejercicio.

Paso 1: Definición de estados y restricciones

```
INIT_STATE: Tuple[int, ...] = (1, 0, 1,
                               0, 0, 0,
                               2, 0, 2)

GOAL_STATE: Tuple[int, ...] = (2, 0, 2,
                               0, 0, 0,
                               1, 0, 1)
```

Donde 0 representa a los lugares donde no existe un caballo, 1 los caballos negros y 2 los caballos blancos. Esto se puede ver de manera más clara a continuación.

```
Índices:      0  1  2        Estado inicial:
             3  4  5        1  0  1   =  ♞  ·  ♞
             6  7  8        0  0  0      ·  ·  ·
                            2  0  2      ♘  ·  ♘
```

La siguiente variable representa los desplazamientos relativos de fila y columna que puede realizar un caballo, representado por la dupla (r,c) donde r es fila y c es columna (row, column).

```
KNIGHT_OFFSETS = [(2, 1), (1, 2), (-1, 2), (-2, 1),
```

```
                 (-2, -1), (-1, -2), (1, -2), (2, -1)]

Movimiento       Descripción
(2, 1)           2 abajo, 1 derecha
(1, 2)           1 abajo, 2 derecha
(-1, 2)          1 arriba, 2 derecha
(-2, 1)          2 arriba, 1 derecha
(-2, -1)         2 arriba, 1 izquierda
(-1, -2)         1 arriba, 2 izquierda
(1, -2)          1 abajo, 2 izquierda
(2, -1)          2 abajo, 1 izquierda
```

Paso 2: Crear diccionario de destinos posibles.

Ahora creamos un diccionario, que nos indique para cada casilla del tablero “¿A qué casillas puedo saltar si hay un caballo aquí?”. Para esto, necesitamos tener funciones que nos permitan pasar de un sistema coordenado a otro, como se ilustra a continuación.

```
Índices (lineales):        Coordenadas (fila, columna):

  0  1  2                    (0,0) (0,1) (0,2)
  3  4  5         <==>       (1,0) (1,1) (1,2)
  6  7  8                    (2,0) (2,1) (2,2)

Funciones auxiliares:

def idx_to_coord(idx: int) -> Tuple[int, int]:
   return divmod(idx, 3)


def coord_to_idx(r: int, c: int) -> int:
   return 3 * r + c
```

Después, con estas funciones auxiliares podemos construir el algoritmo principal, el cual para cada casilla `i` del tablero (0 a 8) convierte primero a coordenadas `(r,c)` usando `idx_to_coord`, luego prueba cada uno de los 8 movimientos posibles de un caballo, desplazando `(r,c)` usando `(dr,dc)` de `KNIGHT_OFFSET`, sí la nueva posición está dentro del tablero, la convierte a coor_to_idx para que sea útil.

Por ejemplo, con `i = 4` (justo en el centro del tablero).

```
r, c = idx_to_coord(4) → (1, 1)

Ahora se prueban los movimientos:
```

```
    (2, 1) → (3, 2) ❌ fuera del tablero

    (1, 2) → (2, 3) ❌ fuera

    (-1, 2) → (0, 3) ❌ fuera

    (-2, 1) → (-1, 2) ❌ fuera

    (-2, -1) → (-1, 0) ❌ fuera

    (-1, -2) → (0, -1) ❌ fuera

    (1, -2) → (2, -1) ❌ fuera

    (2, -1) → (3, 0) ❌ fuera

ENTONCES:
      DEST[4] = []
```

Ya que no existe ningún movimiento válido dentro del tablero, entonces en esa posición, no tenemos ningún movimiento posible.

Por ejemplo, con `i = 0` (esquina superior izquierda).

```
r, c = idx_to_coord(0) → (0, 0)

Ahora se prueban los movimientos:
      (2,1) → (2,1) ✅ → coord_to_idx(2,1) = 7
      (1,2) → (1,2) ✅ → coord_to_idx(1,2) = 5
      los demás están fuera ❌

ENTONCES:
      DEST[0] = [7, 5]

POR LO TANTO:
      DEST = {0: [7, 5],.....4: [],.....}
```

Paso 3: Función generadora de transiciones.

La siguiente función, nos ayuda a calcular los estados vecinos posibles dada una posición actual, haciendo uso del diccionario antes generado.

```
for i, piece in enumerate(s):
   if piece in (1, 2):  # si hay un caballo
       for d in DEST[i]:  # para cada destino posible
```

```
            if s[d] == 0:  # si está vacío
                mover el caballo desde i hasta d
                yield nuevo estado
```

Paso 4: Generar el grafo haciendo uso de Networkx.

El siguiente algoritmo del programa, explora todo el espacio de estados, desde el estado inicial, para realizar nuevos nodos y conexiones dirigidas entre estos. Este espacio de estados puede ser visto posteriormente en una ventana interactiva o como slides en un Jupiternotebook.

```
g = nx.DiGraph()             # grafo dirigido
g.add_node(start)            # nodo inicial
q = deque([start])           # cola para BFS

while q:
   u = q.popleft()          # toma el nodo actual
   for v in successors(u):      # genera estados vecinos
       if v not in g:
           q.append(v)      # nuevos nodos a explorar
       g.add_edge(u, v)     # conexión dirigida u → v
```

Paso 5: Hacer uso de algoritmos de búsqueda.

La función shortest_path() de Networkx hace uso del algoritmo de Búsqueda de Anchura (BFS), el cual sigue caminos desde el nodo inicial hacia sus vecinos (como hacer movimientos con el caballo) y guarda un rastro de cómo llegó a cada nodo. Cuando encuentra el nodo final objetivo, reconstruye el camino que realizó hacia atrás, luego genera el árbol de búsqueda que menos se demoró. El arbol de busqueda principal se puede ver como:

```
Nivel 0: Estado inicial
Nivel 1: Todos los estados alcanzables con 1 movimiento
Nivel 2: Todos los estados alcanzables con 2 movimientos
...
Nivel N: Estado objetivo (¡el primero que lo encuentra es el más corto)
```