Proyecto Final - Análisis de Algoritmos

Alberto Vigna & Camilo Martinez

May 2024

Abstract

Este proyecto aborda la solución del juego Masyu, un rompecabezas de lógica que consiste en conectar todas las perlas en una cuadrícula de $n \times n$ posiciones siguiendo reglas específicas. Se presenta un algoritmo que emplea la heurística de distancia Manhattan y el algoritmo de búsqueda A* para encontrar una solución válida que conecte todas las perlas, respetando las condiciones impuestas por su color (perlas blancas y negras). El algoritmo desarrollado se integra con una interfaz gráfica implementada en Python utilizando la biblioteca Tkinter, permitiendo a los usuarios interactuar con el juego y solicitar la solución automática. Los resultados demuestran que el enfoque propuesto es efectivo para resolver el juego Masyu, proporcionando soluciones óptimas que cumplen con todas las reglas del juego.

1 Análisis

El juego del collar de perlas (Masyu) es un rompecabezas lógico que se juega en una cuadrícula de tamaño $n \times n$. Cada celda de la cuadrícula puede contener una perla blanca, una perla negra o estar vacía. El objetivo del juego es dibujar una única línea continua que pase por todas las perlas respetando ciertas reglas específicas.

1.1 Reglas del Juego

Las reglas del juego Masyu son las siguientes:

1. La línea debe formar un único bucle continuo sin intersecciones ni ramificaciones.
2. La línea debe pasar por todas las perlas exactamente una vez.
3. La línea debe seguir reglas específicas al pasar por perlas blancas y negras.

1.2 Condiciones para las Perlas Blancas

Sea (i, j) la posición de una perla blanca en la cuadrícula. La línea debe cumplir las siguientes condiciones al pasar por una perla blanca:

1. La línea debe pasar recta a través de la perla blanca.
2. La línea debe girar 90 grados inmediatamente antes o después de la perla blanca.

Formalmente, si (i, j) es una perla blanca, entonces debe existir una secuencia de posiciones $(i_1, j_1), (i, j), (i_2, j_2)$ en la línea tal que:

$$\begin{cases} i_1 = i = i_2 \quad \text{y} \quad j_1 \neq j \neq j_2, \quad \text{ó} \\ j_1 = j = j_2 \quad \text{y} \quad i_1 \neq i \neq i_2 \end{cases}$$

Además, debe haber un giro de 90 grados en una de las posiciones adyacentes:

$$\begin{cases} (i_0, j_1) \text{ tal que } (i_0 = i \pm 1 \text{ y } j_1 = j) \quad \text{ó} \\ (i_2, j_3) \text{ tal que } (i_2 = i \text{ y } j_3 = j \pm 1) \end{cases}$$

1.3 Condiciones para las Perlas Negras

Sea (i, j) la posición de una perla negra en la cuadrícula. La línea debe cumplir las siguientes condiciones al pasar por una perla negra:

1. La línea debe girar 90 grados en la perla negra.
2. La línea debe continuar recta en ambas direcciones después del giro.

Formalmente, si (i, j) es una perla negra, entonces debe existir una secuencia de posiciones $(i_1, j_1), (i, j), (i_2, j_2)$ en la línea tal que:

$$\begin{cases} (i_1 = i \text{ y } j_1 = j \pm 1) \quad \text{ó} \\ (i_2 = i \pm 1 \text{ y } j_2 = j) \end{cases}$$

Además, debe haber un giro de 90 grados en la perla negra y continuar recta después del giro:

$$\begin{cases} (i_3 = i \pm 1 \text{ y } j_1 = j) \quad \text{ó} \\ (i_2 = i \text{ y } j_3 = j \pm 1) \end{cases}$$

1.4 Verificación de la Solución

Para verificar que una solución es válida, se debe comprobar que la línea cumple las siguientes condiciones:

- La línea es continua y sin intersecciones.

- La línea pasa por todas las perlas exactamente una vez.
- La línea respeta las reglas al pasar por perlas blancas y negras.

Para formalizar la verificación de la continuidad de la línea, se debe asegurar que para cada par de posiciones consecutivas (i, j) y (i', j') en la línea, se cumple que:

$$|i - i'| + |j - j'| = 1$$

Esta condición asegura que cada par de posiciones consecutivas están adyacentes en la cuadrícula.

Finalmente, para verificar que la solución respeta las reglas de las perlas, se deben aplicar las condiciones matemáticas descritas anteriormente para las perlas blancas y negras a lo largo de toda la línea.

2 Diseño

El diseño del proyecto Masyu se distribuye en varias partes principales: la lectura del archivo de entrada, la implementación de la lógica del juego y la interfaz gráfica de usuario. A continuación, se detallan cada una de estas partes.

2.1 Estructura del Código

El código se divide en los siguientes componentes:

- **Verificaciones**: Conjunto de funciones que se encargan de verificar la ruta trazada y que esta siga las normas del juego
- **Masyu**: Clase principal que maneja la lógica del juego y la interfaz gráfica.
- **supersolver**: Todos los métodos utilizados para poder resolver (o intentar mejor dicho) el tablero.

2.2 Entradas y Salidas del Sistema

2.2.1 Entradas

El sistema recibe una serie de entradas que determinan el estado inicial del tablero de Masyu. Estas entradas se leen desde un archivo de texto y son las siguientes:

1. Una matriz de tamaño $n \times n$ que representa la cuadrícula del juego, donde cada celda puede contener una perla blanca, una perla negra o estar vacía.
2. Una lista de tuplas (i, j, t) donde i y j son las coordenadas de la perla en la matriz y t indica el tipo de perla (1 para blanca y 2 para negra).

Matemáticamente, el archivo de entrada se representa como:

$$\text{entrada} = \begin{cases} n & \text{(número de filas y columnas)} \\ (i_1, j_1, t_1) & \text{(posición y tipo de la primera perla)} \\ (i_2, j_2, t_2) & \text{(posición y tipo de la segunda perla)} \\ \vdots & \vdots \\ (i_k, j_k, t_k) & \text{(posición y tipo de la k-ésima perla)} \end{cases}$$

2.2.2 Salidas

La salida del sistema es una lista de tuplas que indican las posiciones por donde pasa la ruta en el tablero:

$$\text{salida} = [(i_1, j_1), (i_2, j_2), \ldots, (i_m, j_m)]$$

Donde cada (i, j) representa una celda en la cuadrícula a través de la cual pasa la línea.

3 Algoritmos

3.1 Verificación

El archivo *verificaciones.py* contiene funciones clave para verificar la validez de una solución en el juego Masyu. Estas funciones comprueban que la línea que conecta las perlas sigue las reglas del juego y no tiene errores. En esta subsección se muestran cada una de las funcionees, su pseudocódigo y una breve explicación.

- **verificar_linea_continua(linea)**: Esta función verifica si una línea es continua, es decir, si cada punto de la línea está adyacente al siguiente.
- **verificar_perla_blanca(linea, fila, columna)**: Esta función verifica si la línea pasa correctamente por una perla blanca. La línea debe pasar recta por la perla y debe haber un giro de 90 grados inmediatamente antes o después de la perla.
- **verificar_perla_negra(linea, fila, columna)**: Esta función verifica si la línea pasa correctamente por una perla negra. La línea debe girar 90 grados en la perla y continuar recta después del giro.
- **verificar_solucion(linea, perlas)**: Esta función verifica si toda la solución es correcta, comprobando que la línea es continua y que pasa correctamente por todas las perlas según su tipo (blanca o negra).

3.1.1 Pseudocódigo

3.1.2 verificar_linea_continua(linea)

Algorithm 1 verificar_linea_continua

```
1: procedure VERIFICAR_LINEA_CONTINUA(linea)
2:     if longitud de linea ¡ 2 then
3:         return Falso
4:     end if
5:     for i ← 1 to longitud de linea - 1 do
6:         (y1, x1) ← linea[i − 1]
7:         (y2, x2) ← linea[i]
8:         if |y1 − y2| + |x1 − x2| ≠ 1 then
9:             return Falso
10:        end if
11:    end for
12:    return Verdadero
13: end procedure
```

Esta función verifica que cada punto de la línea esté adyacente al siguiente, asegurando que la línea es continua.

3.1.3 verificar_perla_blanca(linea, fila, columna)

Algorithm 2 verificar_perla_blanca

```
1: procedure VERIFICAR_PERLA_BLANCA(linea, fila, columna)
2:     indices ← encontrar todos los índices de linea donde punto == (fila, columna)
3:     if longitud de indices ≠ 1 then
4:         return Falso
5:     end if
6:     indice ← indices[0]
7:     if indice == 0 or indice == longitud de linea - 1 then
8:         return Falso
9:     end if
10:    (y1, x1) ← linea[indice − 1]
11:    (y2, x2) ← linea[indice + 1]
12:    if (y1 == fila and y2 == fila and x1 ≠ columna and x2 ≠ columna) or (x1 == columna and x2 == columna and y1 ≠ fila and y2 ≠ fila) then
13:        return Verdadero
14:    end if
15:    return Falso
16: end procedure
```

Esta función verifica que la línea pasa recta por la perla blanca y que hay un giro de 90 grados inmediatamente antes o después de la perla.

3.1.4 verificar_perla_negra(linea, fila, columna)

Algorithm 3 verificar_perla_negra

```
1: procedure VERIFICAR_PERLA_NEGRA(linea, fila, columna)
2:     indices ← encontrar todos los índices de linea donde punto == (fila, columna)
3:     if longitud de indices ≠ 1 then
4:         return Falso
5:     end if
6:     indice ← indices[0]
7:     if indice == 0 or indice == longitud de linea - 1 then
8:         return Falso
9:     end if
10:    (y1, x1) ← linea[indice − 1]
11:    (y2, x2) ← linea[indice + 1]
12:    if no ((y1 == fila and x2 == columna and |x1 − columna| == 1 and |y2 − fila| == 1) or (x1 == columna and y2 == fila and |y1 − fila| == 1 and |x2 − columna| == 1)) then
13:        return Falso
14:    end if
15:    if indice + 2 ¡ longitud de linea then
16:        (y3, x3) ← linea[indice + 2]
17:        if (y2 == fila and x2 ≠ columna and y3 ≠ fila) or (x2 == columna and y2 ≠ fila and x3 ≠ columna) then
18:            return Falso
19:        end if
20:    end if
21:    return Verdadero
22: end procedure
```

Esta función verifica que la línea gira 90 grados en la perla negra y continúa recta después del giro.

3.1.5 verificar_solucion(linea, perlas)

Algorithm 4 verificar_solucion

```
1: procedure VERIFICAR_SOLUCION(linea, perlas)
2:     errores ← []
3:     if no verificar_linea_continua(linea) then
4:         agregar ("Línea no continua",) a errores
5:     end if
6:     for cada (fila, columna, tipo) en perlas do
7:         if tipo == 1 then
8:             if no verificar_perla_blanca(linea, fila - 1, columna - 1) then
9:                 agregar (fila - 1, columna - 1) a errores
10:            end if
11:        end if
12:        if tipo == 2 then
13:            if no verificar_perla_negra(linea, fila - 1, columna - 1) then
14:                agregar (fila - 1, columna - 1) a errores
15:            end if
16:        end if
17:    end for
18:    return longitud de errores == 0, errores
19: end procedure
```

Esta función verifica que la solución es correcta comprobando que la línea es continua y que pasa correctamente por todas las perlas.

3.1.6 Complejidad de los Algoritmos

verificar_linea_continua(linea): La complejidad es $O(n)$, donde n es la longitud de la línea, ya que revisa cada segmento de la línea una vez.

verificar_perla_blanca(linea, fila, columna): La complejidad es $O(n)$, donde n es la longitud de la línea, ya que recorre la línea para encontrar los índices de la perla y verifica las posiciones adyacentes.

verificar_perla_negra(linea, fila, columna): La complejidad es $O(n)$, donde n es la longitud de la línea, ya que recorre la línea para encontrar los índices de la perla y verifica las posiciones adyacentes.

verificar_solucion(linea, perlas): La complejidad es $O(m \cdot n)$, donde m es el número de perlas y n es la longitud de la línea, ya que verifica cada perla en la línea.

3.2 Solucionador

El archivo *supersolver.py* contiene funciones que utilizan la heurística de la distancia de Manhattan y el algoritmo A* para encontrar y completar una ruta válida en el juego Masyu. La distancia de Manhattan es una medida heurística

utilizada para estimar la distancia entre dos puntos en una cuadrícula, calculada como la suma de las diferencias absolutas de sus coordenadas. El algoritmo A* es un algoritmo de búsqueda que utiliza una función de costo para encontrar la ruta más corta entre dos puntos. A continuación se presenta una descripción detallada de cada función, su pseudocódigo y una breve explicación.

3.2.1 Pseudocódigo

distancia_manhattan(fila1, columna1, fila2, columna2): Esta función calcula la distancia de Manhattan entre dos puntos en la cuadrícula.

Algorithm 5 distancia_manhattan

```
1: procedure DISTANCIA_MANHATTAN(fila1, columna1, fila2, columna2)
2:     return |fila1 − fila2| + |columna1 − columna2|
3: end procedure
```

Esta función retorna la suma de las diferencias absolutas de las coordenadas de dos puntos, proporcionando una estimación de la distancia.

obtener_vecinos(fila, columna, n_filas, n_columnas): Esta función obtiene los vecinos válidos de una posición en la cuadrícula.

Algorithm 6 obtener_vecinos

```
 1: procedure OBTENER_VECINOS(fila, columna, n_filas, n_columnas)
 2:     vecinos ← []
 3:     if fila ¿ 0 then
 4:         vecinos.append(fila - 1, columna)
 5:     end if
 6:     if fila ¡ n_filas - 1 then
 7:         vecinos.append(fila + 1, columna)
 8:     end if
 9:     if columna ¿ 0 then
10:         vecinos.append(fila, columna - 1)
11:     end if
12:     if columna ¡ n_columnas - 1 then
13:         vecinos.append(fila, columna + 1)
14:     end if
15:     return vecinos
16: end procedure
```

Esta función retorna una lista de posiciones adyacentes válidas (vecinos) en la cuadrícula.

encontrar_ruta(inicio, meta, n_filas, n_columnas, ruta_existente): Esta función utiliza el algoritmo A* para encontrar la ruta más corta entre dos puntos específicos en la cuadrícula.

Algorithm 7 encontrar_ruta

```
1: procedure ENCONTRAR_RUTA(inicio, meta, n_filas, n_columnas, ruta_existente)
2:     open_set ← priority_queue()
3:     open_set.push(0, inicio)
4:     came_from ← map()
5:     g_score ← map(inicio → 0)
6:     f_score ← map(inicio → distancia_manhattan(inicio, meta))
7:     while open_set is not empty do
8:         current ← open_set.pop()
9:         if current == meta then
10:            return reconstruir_camino(came_from, current)
11:        end if
12:        for vecino in obtener_vecinos(current, n_filas, n_columnas) do
13:            if vecino in ruta_existente then
14:                continue
15:            end if
16:            tentative_g_score ← g_score[current] + 1
17:            if vecino not in g_score or tentative_g_score ¡ g_score[vecino] then
18:                came_from[vecino] ← current
19:                g_score[vecino] ← tentative_g_score
20:                f_score[vecino] ← tentative_g_score + distancia_manhattan(vecino, meta)
21:                open_set.push(f_score[vecino], vecino)
22:            end if
23:        end for
24:    end while
25:    return None                                    ▷ No se encontró una ruta
26: end procedure
```

Esta función utiliza el algoritmo A* para encontrar la ruta más corta entre dos puntos específicos en la cuadrícula, evitando las posiciones ya existentes en la ruta.

completar_ruta(n_filas, n_columnas, perlas, ruta_actual): Esta función completa la ruta pasando por todas las perlas en la cuadrícula.

Algorithm 8 completar_ruta

```
1: procedure COMPLETAR_RUTA(n_filas, n_columnas, perlas, ruta_actual)
2:     if ruta_actual is empty then
3:         return []
4:     end if
5:     puntos_clave ← ruta_actual.copy()
6:     perlas_ordenadas ← sorted(perlas, key = lambda p:
   distancia_manhattan(p[0], p[1], ruta_actual[-1][0], ruta_actual[-1][1]))
7:     puntos_clave.extend([(p[0] - 1, p[1] - 1) for p in perlas_ordenadas if (p[0]
   - 1, p[1] - 1) not in puntos_clave])
8:     puntos_clave.append(ruta_actual[0])
9:     ruta_completa ← []
10:    for i in range(len(puntos_clave) - 1) do
11:        parte_ruta ← encontrar_ruta(puntos_clave[i], puntos_clave[i + 1],
   n_filas, n_columnas, ruta_completa)
12:        if parte_ruta then
13:            ruta_completa.extend(parte_ruta[1:])
14:        else
15:            return ruta_completa
16:        end if
17:    end for
18:    return ruta_completa
19: end procedure
```

Esta función utiliza la función **encontrar_ruta** para conectar todos los puntos clave (perlas y el punto de inicio) en la cuadrícula, completando la ruta.

reconstruir_camino(came_from, actual): Esta función reconstruye el camino desde el punto final hasta el punto inicial usando el diccionario de seguimiento.

Algorithm 9 reconstruir_camino

```
1: procedure RECONSTRUIR_CAMINO(came_from, actual)
2:     total_path ← [actual]
3:     while actual in came_from do
4:         actual ← came_from[actual]
5:         total_path.append(actual)
6:     end while
7:     return total_path[::-1]
8: end procedure
```

Esta función utiliza el diccionario de seguimiento **came_from** para reconstruir la ruta desde el punto final hasta el inicial.

3.2.2 Complejidad de los Algoritmos

distancia_manhattan(fila1, columna1, fila2, columna2): La complejidad es $O(1)$ ya que solo realiza operaciones aritméticas simples.

obtener_vecinos(fila, columna, n_filas, n_columnas): La complejidad es $O(1)$ ya que el número de vecinos es constante y no depende del tamaño de la cuadrícula.

encontrar_ruta(inicio, meta, n_filas, n_columnas, ruta_existente): La complejidad en el peor caso es $O(b^d)$, donde b es el factor de ramificación y d es la profundidad de la solución en el espacio de búsqueda. En el mejor caso, la complejidad es $O(n \log n)$, siendo n el número de nodos explorados.

completar_ruta(n_filas, n_columnas, perlas, ruta_actual): La complejidad depende del número de perlas y la longitud de la ruta. En el peor caso, si hay m perlas, la complejidad es $O(m \cdot b^d)$.

reconstruir_camino(came_from, actual): La complejidad es $O(n)$, donde n es la longitud del camino, ya que reconstruye la ruta recorriendo el diccionario **came_from**.

3.3 Masyu

El archivo principal del proyecto, *masyu.py*, implementa la lógica del juego Masyu y la interfaz gráfica de usuario utilizando la biblioteca Tkinter. Este archivo contiene funciones para leer el archivo de entrada, manejar los eventos del usuario, dibujar el tablero y las perlas, así como verificar y completar la ruta automáticamente.

leer_archivo_entrada(ruta): Esta función lee el archivo de entrada y extrae la configuración del tablero, incluyendo el tamaño del tablero y la posición de las perlas.

Algorithm 10 leer_archivo_entrada

```
1:  procedure LEER_ARCHIVO_ENTRADA(ruta)
2:      file ← abrir(ruta)
3:      lineas ← file.readlines()
4:      n ← int(lineas[0].strip())
5:      perlas ← []
6:      for linea in lineas[1:] do
7:          fila, columna, tipo ← map(int, linea.strip().split(','))
8:          perlas.append((fila, columna, tipo))
9:      end for
10:     return n, n, perlas
11: end procedure
```

Esta función retorna el tamaño del tablero y una lista de perlas con sus posiciones y tipos.

Clase Masyu: La clase principal que maneja la lógica del juego y la interfaz gráfica. Contiene métodos para dibujar el tablero, dibujar perlas, manejar

eventos de usuario y verificar o completar la ruta.

Algorithm 11 Masyu: __init__

```
1: procedure __INIT__(master, n_filas, n_columnas, perlas)
2:     self.master ← master
3:     self.n_filas ← n_filas
4:     self.n_columnas ← n_columnas
5:     self.perlas ← perlas
6:     self.tablero ← [[0 for _ in range(n_columnas)] for _ in range(n_filas)]
7:     self.canvas ← tk.Canvas(master, width=n_columnas * 40, height=n_filas
   * 40)
8:     self.canvas.pack()
9:     self.dibujar_tablero()
10:    self.dibujar_perlas()
11:    self.linea_actual ← []
12:    self.lineas_dibujadas ← []
13:    self.canvas.bind("¡Button-1¿", self.iniciar_linea)
14:    master.bind("¡Up¿", self.mover_arriba)
15:    master.bind("¡Down¿", self.mover_abajo)
16:    master.bind("¡Left¿", self.mover_izquierda)
17:    master.bind("¡Right¿", self.mover_derecha)
18:    self.boton_verificar ← tk.Button(master, text="Verificar Solución",
   command=self.verificar_solucion)
19:    self.boton_verificar.pack()
20:    self.boton_auto ← tk.Button(master, text="Completar Ruta",
   command=self.completar_ruta_automatica)
21:    self.boton_auto.pack()
22: end procedure
```

Este constructor inicializa el tablero y la interfaz gráfica, y configura los eventos de usuario.

completar_ruta_automatica(): Este método completa automáticamente la ruta utilizando la función **completar_ruta** del archivo *supersolver.py*.

Algorithm 12 completar_ruta_automatica

```
1: procedure COMPLETAR_RUTA_AUTOMATICA
2:     nueva_ruta ← completar_ruta(self.n_filas, self.n_columnas, self.perlas, self.linea_actual)
3:     if nueva_ruta then
4:         self.linea_actual ← nueva_ruta
5:         self.dibujar_linea()
6:         self.imprimir_ruta()
7:     else
8:         messagebox.showinfo("Resultado", "No se encontró una ruta completa, mostrando la mejor ruta encontrada.")
9:         self.dibujar_linea()
10:        self.imprimir_ruta()
11:    end if
12: end procedure
```

Este método llama a la función **completar_ruta** para intentar completar la ruta y actualiza la interfaz gráfica con la ruta encontrada.

dibujar_tablero(): Este método dibuja la cuadrícula del tablero en el canvas de Tkinter.

Algorithm 13 dibujar_tablero

```
1: procedure DIBUJAR_TABLERO
2:     for i in range(self.n_filas) do
3:         for j in range(self.n_columnas) do
4:             self.canvas.create_rectangle(j * 40, i * 40, (j + 1) * 40, (i + 1) * 40, outline="black")
5:         end for
6:     end for
7: end procedure
```

Este método crea un rectángulo para cada celda de la cuadrícula.

dibujar_perlas(): Este método dibuja las perlas en la cuadrícula basándose en sus posiciones y tipos.

Algorithm 14 dibujar_perlas

```
1: procedure DIBUJAR_PERLAS
2:     for fila, columna, tipo in self.perlas do
3:         x, y ← (columna - 1) * 40 + 20, (fila - 1) * 40 + 20
4:         color ← 'white' if tipo == 1 else 'black'
5:         self.canvas.create_oval(x - 15, y - 15, x + 15, y + 15, fill=color, outline="black")
6:     end for
7: end procedure
```

Este método crea un óvalo para cada perla en sus respectivas posiciones.

iniciar_linea(event): Este método inicia una nueva línea en la posición donde el usuario hace clic.

Algorithm 15 iniciar_linea

```
1: procedure INICIAR_LINEA(event)
2:     self.linea_actual ← [(event.y // 40, event.x // 40)]
3:     self.dibujar_linea()
4:     self.imprimir_ruta()
5: end procedure
```

Este método inicia una nueva línea y la dibuja en el canvas.

mover_arriba(event), **mover_abajo(event)**, **mover_izquierda(event)**, **mover_derecha(event)**: Estos métodos permiten mover la línea en las cuatro direcciones cardinales.

Algorithm 16 mover_arriba

```
1: procedure MOVER_ARRIBA(event)
2:     if self.linea_actual then
3:         y, x ← self.linea_actual[-1]
4:         if y ¿ 0 then
5:             self.linea_actual.append((y - 1, x))
6:             self.dibujar_linea()
7:             self.imprimir_ruta()
8:         end if
9:     end if
10: end procedure
```

Algorithm 17 mover_abajo

```
1: procedure MOVER_ABAJO(event)
2:     if self.linea_actual then
3:         y, x ← self.linea_actual[-1]
4:         if y ¡ self.n_filas - 1 then
5:             self.linea_actual.append((y + 1, x))
6:             self.dibujar_linea()
7:             self.imprimir_ruta()
8:         end if
9:     end if
10: end procedure
```

Algorithm 18 mover_izquierda

```
1: procedure MOVER_IZQUIERDA(event)
2:     if self.linea_actual then
3:         y, x ← self.linea_actual[-1]
4:         if x ¿ 0 then
5:             self.linea_actual.append((y, x - 1))
6:             self.dibujar_linea()
7:             self.imprimir_ruta()
8:         end if
9:     end if
10: end procedure
```

Algorithm 19 mover_derecha

```
1: procedure MOVER_DERECHA(event)
2:     if self.linea_actual then
3:         y, x ← self.linea_actual[-1]
4:         if x ¡ self.n_columnas - 1 then
5:             self.linea_actual.append((y, x + 1))
6:             self.dibujar_linea()
7:             self.imprimir_ruta()
8:         end if
9:     end if
10: end procedure
```

Estos métodos permiten al usuario extender la línea en cualquier dirección usando las teclas de flecha.

dibujar_linea(): Este método dibuja la línea actual en el canvas.

Algorithm 20 dibujar_linea

```
1: procedure DIBUJAR_LINEA
2:     self.canvas.delete("linea")
3:     for i in range(len(self.linea_actual) - 1) do
4:         y1, x1 ← self.linea_actual[i]
5:         y2, x2 ← self.linea_actual[i + 1]
6:         self.canvas.create_line(x1 * 40 + 20, y1 * 40 + 20, x2 * 40 + 20, y2
   * 40 + 20, fill="red", width=2, tag="linea")
7:     end for
8: end procedure
```

Este método dibuja una línea roja en el canvas entre los puntos especificados en **linea_actual**.

imprimir_ruta(): Este método imprime la ruta actual en la consola.

Algorithm 21 imprimir_ruta

```
1: procedure IMPRIMIR_RUTA
2:     print("Ruta actual:", self.linea_actual)
3: end procedure
```

Este método simplemente imprime la lista de puntos por los que pasa la línea actual.

verificar_solucion(): Este método verifica si la solución actual es correcta utilizando la función **verificar_solucion** del archivo *verificaciones.py*.

Algorithm 22 verificar_solucion

```
1: procedure VERIFICAR_SOLUCION
2:     es_correcto, errores ← verificar_solucion(self.linea_actual, self.perlas)
3:     if es_correcto then
4:         messagebox.showinfo("Resultado", "¡Solución correcta!")
5:     else
6:         mensaje_error ← "Solución incorrecta. Errores en las celdas:" +
   "".join([f"Fila e[0] + 1, Columna e[1] + 1" for e in errores])
7:         messagebox.showerror("Resultado", mensaje_error)
8:         print("Errores en las celdas:", errores)
9:     end if
10: end procedure
```

Este método muestra un mensaje indicando si la solución es correcta o incorrecta, y lista los errores si los hay.

3.3.1 Complejidad de los Algoritmos

leer_archivo_entrada(ruta): La complejidad es $O(n)$, donde n es el número de líneas en el archivo, ya que lee y procesa cada línea una vez.

Masyu: __init__ y métodos de dibujo: La complejidad de los métodos de dibujo es $O(n \times m)$, donde n es el número de filas y m es el número de columnas, ya que dibuja cada celda del tablero y cada perla una vez.

completar_ruta_automatica(), **verificar_solucion()**: Estas funciones dependen de las funciones **completar_ruta** y **verificar_solucion** respectivamente, cuyas complejidades ya han sido analizadas.