# Proyecto - Masyu

Alejandro Salamanca, Andrés Salamanca

29 de mayo de 2024

**Abstract**

El documento presenta un análisis y propuesta de dos algoritmos para resolver el juego de Masyu, un problema NP completo. Los algoritmos discutidos son el método de intersección y la búsqueda por eliminación. Se aborda la complejidad del juego y se ofrece una solución estructurada que involucra el diseño de los algoritmos, su pseudocódigo y un análisis de su complejidad. Este trabajo busca proporcionar una comprensión más profunda de los desafíos algorítmicos asociados con el juego de Masyu y ofrecer herramientas para abordarlos eficientemente.

# Part I
# Análisis y diseño del problema

## 1 Análisis

El análisis del problema del juego Masyu revela su naturaleza como un problema SAT (Satisfacción de Restricciones Booleanas). En este juego, se presenta un tablero de tamaño$NxN$ con perlas dispuestas de diversas maneras. Estas perlas imponen restricciones específicas: las perlas negras requieren que una línea pase a través de ellas y gire inmediatamente, con una línea recta continuando después del giro; mientras que las perlas blancas deben tener una línea recta pasando a través de ellas, con al menos un giro en los espacios adyacentes. El objetivo es formar un collar de perlas, creando un ciclo que cumpla con todas las restricciones impuestas por las perlas. Este análisis proporciona una comprensión inicial de la complejidad del problema y su relación con la teoría SAT, lo que orienta hacia posibles enfoques algorítmicos para resolverlo.

El problema del juego Masyu puede representarse de manera efectiva utilizando la teoría de grafos, lo que proporciona una perspectiva estructurada para abordar las restricciones y objetivos del juego. En esta representación, el tablero de tamaño $N \times N$ se modela como un grafo $G(V, E)$, donde cada celda del tablero corresponde a un vértice $V$ y las conexiones entre celdas adyacentes (verticales y horizontales) se representan mediante aristas $E$.

En este grafo, cada vértice $V$ puede tener un máximo de cuatro aristas, que representan las conexiones posibles con sus celdas adyacentes: arriba, abajo, izquierda y derecha. Las conexiones diagonales no están permitidas, lo que añade una restricción adicional en la estructura del grafo. Esto significa que cada vértice tiene como máximo cuatro vecinos, limitando así el grado del vértice a cuatro.

Los vértices del grafo pueden representar tres tipos diferentes de celdas en el tablero de Masyu: espacios en blanco, perlas blancas y perlas negras. Estas se codifican como sigue:

- Los espacios en blanco se representan con el valor 0.
- Las perlas blancas se representan con el valor 1 .
- Las perlas negras se representan con el valor 2 .

Cada tipo de perla impone restricciones específicas sobre cómo las líneas pueden pasar a través de ellas. En el caso de las perlas negras, cualquier línea que pase por una de estas debe hacer un giro en la celda y continuar en línea recta después del giro. Para las perlas blancas, la línea debe pasar recta a través de la celda, pero debe girar al menos una vez en las celdas adyacentes. Estas restricciones pueden traducirse en condiciones que las aristas del grafo deben satisfacer, lo que complica la búsqueda de un ciclo válido en el grafo.

## 1.1 Restricciones SAT

Las restricciones SAT del problema de Masyu están directamente relacionadas con las perlas negras y blancas en el tablero. Para cada perla negra $(i, j)$, si una línea pasa por esta celda, la línea debe girar en $(i, j)$ y continuar en línea recta después del giro. Esto impone una restricción que asegura que la dirección de la línea cambia en la perla negra, y después del giro, la línea sigue una trayectoria recta al menos una celda más allá.

En el caso de las perlas blancas $(i, j)$, la línea debe pasar recta a través de la celda $(i, j)$. Sin embargo, además de pasar de manera recta, debe haber al menos un giro en uno de los espacios adyacentes a la perla blanca. Esto significa que, aunque la línea mantenga una dirección recta en la celda de la perla blanca, debe cambiar de dirección en al menos una de las celdas vecinas, asegurando que el recorrido no sea completamente lineal.

## 1.2 Ejemplo de Restricciones

Perla Negra: Consideremos una perla negra ubicada en $(i, j)$. Si una línea viene desde $(i-1, j)$ y sale hacia$(i, j+1)$, esto significa que la línea debe girar en la celda $(i, j)$. En este caso, las celdas $(i-1, j)$ y $(i, j+1)$ deben estar conectadas por una línea recta antes y después del giro. Esto asegura que la línea hace un giro de 90 grados en la perla negra y sigue una trayectoria recta después del giro, cumpliendo con las restricciones impuestas por este tipo de perla.

Perla Blanca :Para una perla blanca ubicada en $(i, j)$, si la línea pasa de $(i-1, j)$ a $(i+1, j)$, la restricción adicional es que debe haber al menos un giro en los espacios adyacentes a $(i, j)$. Esto significa que debe existir un cambio de dirección en al menos una de las celdas$(i-1, j-1)$, $(i-1, j+1)$, $(i+1, j-1)$ o $(i+1, j+1)$. Esto asegura que aunque la línea pase recta a través de la perla blanca, hay un giro en uno de los espacios vecinos, cumpliendo así las condiciones necesarias para las perlas blancas.

# 2 Diseño

Con las observaciones presentadas en el análisis anterior, podemos escribir el diseño de un algoritmo que solucione el problema . A veces este diseño se conoce como el «contrato» del algoritmmos o las «precondiciones» y «poscondiciones» del algoritmo. El diseño se compone de entradas y salidas:

**Definition.** Entradas:

1. $G(V, E)$: Un grafo donde:
2. $V$: Conjunto de vértices que representan las celdas del tablero. Cada vértice puede ser un espacio en blanco (0), una perla blanca (1) o una perla negra (2).
3. $E$: Conjunto de aristas que representan los caminos posibles entre celdas adyacentes (no diagonales) en el tablero.

**Definition.** Salidas:

1. e*: Un subconjunto de aristas $e* \subseteq E$ que forman un ciclo en el grafo $G$. Este ciclo debe cumplir con todas las restricciones especificadas para las perlas negras y blancas:
2. Para las perlas negras: La línea debe girar en la perla y continuar en línea recta después del giro.
3. Para las perlas blancas: La línea debe pasar recta a través de la perla y debe haber al menos un giro en los espacios adyacentes.

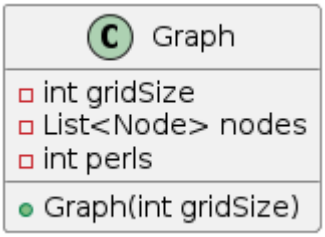


Figura 3.1: Grafo

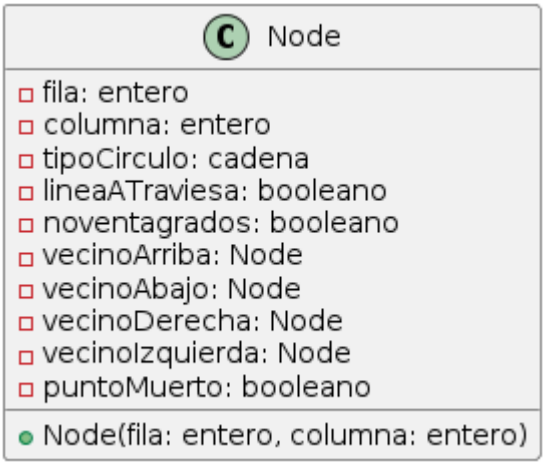


Figura 3.2: Nodo

# Part II
# Algoritmos

## 3 Estructuras de datos

### 3.1 Clases

La clase Graph representa un grafo que modela el tablero del juego Masyu. Posee tres atributos principales: gridSize, que indica el tamaño del tablero; nodes, una lista que almacena los nodos del grafo; y perls, que cuenta el número de perlas presentes en el tablero. El constructor Graph(int gridSize) inicializa la clase, estableciendo el tamaño del tablero, creando una lista vacía para almacenar los nodos y configurando el contador de perlas en cero. Esta clase es fundamental para estructurar y manipular el grafo del juego Masyu.

La clase Node representa un nodo en el grafo del juego Masyu. Cada nodo tiene una fila y una columna que indican su posición en el tablero, un tipo de círculo (circleType) que puede ser 'negro', 'blanco' o nulo si no hay círculo, y un booleano (lineThrough) que indica si una línea pasa a través del nodo. Además, posee un atributo booleano (ninetydegree) que indica si el giro en el nodo es de 90 grados. Los atributos vecinoArriba, vecinoAbajo, vecinoDerecha y vecinoIzquierda son referencias a los nodos vecinos en las cuatro direcciones cardinales, permitiendo la navegación a través del grafo. Finalmente, deadSpot es un booleano que indica si el nodo es un "punto muerto" donde ninguna línea puede pasar a través de él. Esta clase es esencial para representar los nodos del tablero y gestionar sus propiedades en el contexto del juego Masyu.

# 4 Algoritmos utilizados

## 4.1 Algoritmos para calcular la información basica

**Algorithm 1** getWhiteCirclesAtEdge()

```
1: procedure GETWHITECIRCLESATEDGE
2:     whiteCircles ← []
3:     for fila ← 0 to gridSize − 1 do
4:         for columna ← 0 to gridSize − 1 do
5:             if (fila = 0 ∨ fila = gridSize − 1 ∨ columna = 0 ∨ columna =
   gridSize − 1) ∧ getNode(fila, columna).circleType = 1 then
6:                 whiteCircles.append(getNode(fila, columna))
7:             end if
8:         end for
9:     end for
10:    return whiteCircles
11: end procedure
```

El algoritmo getWhiteCirclesAtEdge [1] busca y devuelve una lista de nodos blancos que se encuentran en el borde del tablero. Para hacer esto, el algoritmo itera sobre todas las filas y columnas del tablero utilizando dos bucles for. En cada iteración, verifica si el nodo actual se encuentra en el borde del tablero y si su tipo de círculo es blanco. Si ambas condiciones se cumplen, el nodo se agrega a la lista whiteCircles. Al final, el algoritmo devuelve la lista de nodos blancos encontrados en el borde del tablero.

El algoritmo getBlackCirclesAtEdge [2] es similar al anterior, pero busca nodos negros en el borde del tablero. También utiliza dos bucles for para iterar sobre todas las filas y columnas del tablero. Para cada nodo, verifica si está en el borde del tablero y si su tipo de círculo es negro. Si ambas condiciones se cumplen, el nodo junto con su borde y esquina correspondientes se agrega a la lista blackCircles. Al final, el algoritmo devuelve la lista de nodos negros encontrados en el borde del tablero.

**Algorithm 2** getBlackCirclesAtEdge()

```
1: procedure GETBLACKCIRCLESATEDGE
2:     blackCircles ← []
3:     for fila ← 0 to gridSize − 1 do
4:         for columna ← 0 to gridSize − 1 do
5:             if (fila = 0 ∨ fila = gridSize − 1 ∨ columna = 0 ∨ columna = gridSize − 1) ∧ getNode(fila, columna).circleType = 2 then
6:                 blackCircles.append({node : getNode(fila, columna), edge : getEdge(fila, columna), corner : getCorner(fila, columna)})
7:             end if
8:         end for
9:     end for
10:    return blackCircles
11: end procedure
```

**Algorithm 3** getBlackCirclesOneSpaceFromEdge()

```
1: procedure GETBLACKCIRCLESONESPACEFROMEDGE
2:     blackCircles ← []
3:     for fila ← 1 to gridSize − 2 do
4:         for columna ← 1 to gridSize − 2 do
5:             if (fila = 1 ∨ fila = gridSize − 2 ∨ columna = 1 ∨ columna = gridSize − 2) ∧ getNode(fila, columna).circleType = 2 then
6:                 blackCircles.append({node : getNode(fila, columna), edge : getEdge(fila, columna), corner : getCornerOneSpace(fila, columna)})
7:             end if
8:         end for
9:     end for
10:    return blackCircles
11: end procedure
```

El algoritmo getBlackCirclesOneSpaceFromEdge [3] busca y devuelve una lista de nodos negros que se encuentran a una casilla de distancia del borde del tablero. Utiliza dos bucles for para iterar sobre todas las filas y columnas del tablero, excluyendo el borde. Para cada nodo, verifica si está a una casilla de distancia del borde del tablero y si su tipo de círculo es negro. Si ambas condiciones se cumplen, el nodo junto con su borde y esquina correspondientes se agrega a la lista blackCircles. Al final, el algoritmo devuelve la lista de nodos negros encontrados a una casilla de distancia del borde del tablero.

El algoritmo getThreeOrMoreWhiteCirclesInLine [4] busca y devuelve una lista de secuencias de nodos blancos de longitud 3 o más en línea, ya sea horizontal o verticalmente. Itera a través de todas las filas y columnas del tablero, verificando si hay secuencias de nodos blancos consecutivos en cada fila y columna. Cuando encuentra una secuencia de al menos 3 nodos blancos, la agrega a la lista whiteCircles, junto con la dirección de la secuencia (horizontal o vertical). Finalmente, devuelve la lista de secuencias encontradas.

El algoritmo getTwoAdjacentBlackeCircles [5] busca y devuelve una lista de pares de nodos negros adyacentes, ya sea horizontal o verticalmente. Itera a través de todas las filas y columnas del tablero, verificando si hay pares de nodos negros adyacentes en cada posición. Cuando encuentra un par de nodos negros adyacentes, los agrega a la lista adjacentBlackCircles, junto con la dirección de la adyacencia (horizontal o vertical). Finalmente, devuelve la lista de pares de nodos negros adyacentes encontrados.

El algoritmo getBlackCircleWithWhiteCorners [6] busca y devuelve una lista de nodos negros que tienen esquinas blancas adyacentes. Itera a través de todas las filas y columnas del tablero, verificando si cada nodo es de tipo negro. Luego, verifica si las esquinas superiores izquierda y derecha o las esquinas inferiores izquierda y derecha del nodo son blancas. Si se encuentran esquinas blancas adyacentes, se agrega el nodo junto con las esquinas blancas y la dirección al cual pertenece la esquina en la lista blackCirclesWithWhiteCorners. Finalmente, se devuelve la lista de nodos negros con esquinas blancas adyacentes encontrados.

Los algoritmos del 1 al 6 proporcionan un reconocimiento básico del tablero de juego en el juego de Masyu. Estos algoritmos identifican perlas clave en el tablero que, dependiendo de su posición, pueden tener movimientos específicos. Por ejemplo, si una perla negra está en la esquina del tablero, solo puede tener sus movimientos en dos direcciones.

Estos algoritmos son útiles para el juego de Masyu por varias razones:

- Identificación de perlas clave: Los algoritmos ayudan a identificar perlas negras y blancas en posiciones estratégicas del tablero, como en los bordes o en esquinas. Esto permite al jugador prever movimientos futuros y planificar en consecuencia.

- Análisis de movimientos posibles: Al reconocer patrones específicos de perlas, como las perlas negras con dos movimientos posibles, los algoritmos proporcionan información crucial sobre las posibles jugadas disponibles en cada turno.

**Algorithm 4** getThreeOrMoreWhiteCirclesInLine()

```
procedure GETTHREEORMOREWHITECIRCLESINLINE
    whiteCircles ← []
    for fila ← 0 to gridSize − 1 do
        count ← 0
        sequence ← []
        for col ← 0 to gridSize − 1 do
            node ← getNode(fila, col)
            if node and node.circleType = 1 then
                count ← count + 1
                sequence.append(node)
                if count ≥ 3 then
                    whiteCircles.append({nodes : copy(sequence), direction :
'horizontal'})
                end if
            else
                count ← 0
                sequence ← []
            end if
        end for
    end for
    for col ← 0 to gridSize − 1 do
        count ← 0
        sequence ← []
        for fila ← 0 to gridSize − 1 do
            node ← getNode(fila, col)
            if node and node.circleType = 1 then
                count ← count + 1
                sequence.append(node)
                if count ≥ 3 then
                    whiteCircles.append({nodes : copy(sequence), direction :
'vertical'})
                end if
            else
                count ← 0
                sequence ← []
            end if
        end for
    end for
    return whiteCircles
end procedure
```

**Algorithm 5** getTwoAdjacentBlackeCircles()

```
1: procedure GETTWOADJACENTBLACKECIRCLES
2:     adjacentBlackCircles ← []
3:     for fila ← 0 to gridSize − 1 do
4:         for col ← 0 to gridSize − 1 do
5:             node ← getNode(fila, col)
6:             if node and node.circleType = 2 then
7:                 rightNode ← getNode(fila, col + 1)
8:                 if rightNode and rightNode.circleType = 2 then
9:                     adjacentBlackCircles.append({nodes :
   [node, rightNode], direction : 'horizontal'})
10:                end if
11:                bottomNode ← getNode(fila + 1, col)
12:                if bottomNode and bottomNode.circleType = 2 then
13:                    adjacentBlackCircles.append({nodes :
   [node, bottomNode], direction : 'vertical'})
14:                end if
15:            end if
16:        end for
17:    end for
18:    return adjacentBlackCircles
19: end procedure
```

- Mejora de la precisión en las jugadas: Al reconocer las configuraciones clave del tablero, los algoritmos ayudan al jugador a realizar jugadas más precisas y acertadas, evitando errores comunes y maximizando las oportunidades estratégicas.

## 4.2 Complejidad

Todos los algoritmos presentados tienen una complejidad de tiempo cuadrática $O(N^2)$, lo que significa que el tiempo de ejecución aumenta cuadráticamente con el tamaño del tablero $N$. Esto se debe a que cada algoritmo itera sobre todas las celdas del tablero una vez, realizando operaciones constantes en cada iteración.

## 4.3 Algoritmos para calcular jugadas perlas negras

El algoritmo generateBlackCircleMoves [7] genera y ejecuta movimientos posibles para las perlas negras en el juego de Masyu. Itera sobre todas las celdas del tablero y para cada celda que contiene una perla negra, verifica si ya se ha girado 90 grados. Si la perla negra no se ha girado aún, genera posibles movimientos y verifica si hay un movimiento válido. Si hay exactamente un movimiento válido, lo realiza y lo registra. Finalmente, devuelve la lista de movimientos realizados.

**Algorithm 6** getBlackCircleWithWhiteCorners()

```
1: procedure GETBLACKCIRCLEWITHWHITECORNERS
2:     blackCirclesWithWhiteCorners ← []
3:     for fila ← 0 to gridSize − 1 do
4:         for col ← 0 to gridSize − 1 do
5:             node ← getNode(fila, col)
6:             if node and node.circleType = 2 then
7:                 topLeft ← getNode(fila − 1, col − 1)
8:                 topRight ← getNode(fila − 1, col + 1)
9:                 if topLeft and topLeft.circleType = 1 and topRight and
   topRight.circleType = 1 then
10:                    blackCirclesWithWhiteCorners.append({node, corners :
   [topLeft, topRight], direction : 'top'})
11:                end if
12:                bottomRight ← getNode(fila + 1, col + 1)
13:                bottomLeft ← getNode(fila + 1, col − 1)
14:                if bottomRight and bottomRight.circleType = 1 and
   bottomLeft and bottomLeft.circleType = 1 then
15:                    blackCirclesWithWhiteCorners.append({node, corners :
   [bottomRight, bottomLeft], direction : 'bottom'})
16:                end if
17:            end if
18:        end for
19:    end for
20:    return blackCirclesWithWhiteCorners
21: end procedure
```

**Algorithm 7** generateBlackCircleMoves()

```
1: procedure GENERATEBLACKCIRCLEMOVES
2:     jugadasPosiblesBlack ← []
3:     jugadasHechas ← []
4:     for fila ← 0 to gridSize − 1 do
5:         for col ← 0 to gridSize − 1 do
6:             node ← getNode(fila, col)
7:             if node and node.circleType = 2 then
8:                 setninetydegree(node)
9:                 if ¬node.ninetydegree then
10:                    jugadasPosiblesBlack ← generateMovesFromBlackCircle(fila, col, node)
11:                    filtroJugadas ← checkJugadaBlack(node, jugadasPosiblesBlack)
12:                    if (|filtroJugadas|) = 1 then
13:                        jugar ← filtroJugadas.pop()
14:                        nodeSource ← jugar.primero
15:                        nodeDest ← jugar.segundo
16:                        jugadasHechas.push(jugar)
17:                    end if
18:                end if
19:            end if
20:        end for
21:    end for
22:    return jugadasHechas
23: end procedure
```

**Algorithm 8** generateMovesFromBlackCircle(row, col, node)

```
1:  procedure GENERATEMOVESFROMBLACKCIRCLE(row, col, node)
2:      moves ← []
3:      tot_vec ← num_vecinos(node)
4:      adj ← getAdjacentsall(row, col)
5:      lineThrough ← false
6:      if node.vecinoabajo ≠ null then
7:          lineThrough ← checkLineThrough(node.vecinoabajo)
8:          for a in adj do
9:              if a = null then
10:                 continue
11:             end if
12:             if a.row = node.row + 1 then
13:                 nodetwo ← getNode(a.row + 1, a.col)
14:                 moves.append({primero : a, segundo : nodetwo})
15:             else if a.row = node.row − 1 then
16:                 nodetwo ← getNode(a.row − 1, a.col)
17:                 moves.append({primero : a, segundo : nodetwo})
18:             end if
19:         end for
20:     end if
21:     if node.vecinoarriba ≠ null then
22:         lineThrough ← checkLineThrough(node.vecinoarriba)
23:         for a in adj do
24:             if a = null then
25:                 continue
26:             end if
27:             if a.col = node.col + 1 then
28:                 nodetwo ← getNode(a.row, a.col + 1)
29:                 moves.append({primero : a, segundo : nodetwo})
30:             else if a.col = node.col − 1 then
31:                 nodetwo ← getNode(a.row, a.col − 1)
32:                 moves.append({primero : a, segundo : nodetwo})
33:             end if
34:         end for
35:     end if
36:     if node.vecinoderecha ≠ null then
37:         lineThrough ← checkLineThrough(node.vecinoderecha)
38:         for a in adj do
39:             if a = null then
40:                 continue
41:             end if
42:             if a.row = node.row + 1 then
43:                 nodetwo ← getNode(a.row + 1, a.col)
44:                 moves.append({primero : a, segundo : nodetwo})
45:             else if a.row = node.row − 1 then
46:                 nodetwo ← getNode(a.row − 1, a.col)
47:                 moves.append({primero : a, segundo : nodetwo})
48:             end if
49:         end for
50:     end if
51:     if node.vecinoizquierda ≠ null then
52:         lineThrough ← checkLineThrough(node.vecinoizquierda)
53:         for a in adj do
54:             if a = null then
55:                 continue
```

El algoritmo generateMovesFromBlackCircle(row, col, node) [8] genera movimientos posibles desde una perla negra en el juego de Masyu. Itera sobre los vecinos de la perla negra y verifica si es posible hacer un movimiento válido desde esa posición. Luego, agrega los movimientos posibles a la lista moves y la devuelve al finalizar.

Los algoritmos generateBlackCircleMoves() y generateMovesFromBlackCircle(row, col, node) son fundamentales para la lógica de juego del Masyu.

El primero, generateBlackCircleMoves(), recorre todas las perlas negras en el tablero y genera posibles movimientos a partir de cada una. Utiliza una serie de funciones auxiliares para determinar la viabilidad de cada movimiento, como verificar si la perla negra ya ha sido girada en 90 grados y si hay una línea a través de los vecinos de la perla negra. Finalmente, devuelve una lista de los movimientos válidos realizados.

Por otro lado, generateMovesFromBlackCircle(row, col, node) se encarga de generar movimientos posibles desde una perla negra específica en el tablero. Primero, calcula los vecinos de la perla negra en cuestión y verifica si hay una línea a través de estos vecinos. Luego, genera movimientos válidos basados en la disposición de los vecinos y las reglas del juego. Retorna una lista de los movimientos posibles desde la perla negra dada.

## 4.4 Complejidad

La complejidad de los algoritmos implementados para la generación de movimientos en el juego Masyu es cuadrática,$O(N^2)$, donde $N$ representa el tamaño del tablero. Tanto el algoritmo generateBlackCircleMoves() como generateMovesFromBlackCircle(row, col, node) recorren el tablero y realizan operaciones sobre cada celda y sus vecinos.

## 4.5 Algoritmos para calcular jugadas perlas blancas

El algoritmo, denominado CompleteWhiteCircle [9], se encarga de identificar y completar las líneas que atraviesan perlas blancas con un solo vecino. Su funcionamiento es iterar sobre todas las celdas del tablero, detectando perlas blancas con solo un vecino adyacente. Luego, invoca la función completeLineThrough(node) para determinar la línea adicional necesaria para completar la perla blanca. Posteriormente, realiza la jugada correspondiente, marcando la perla blanca como completada, y añade la jugada a una lista de posibles movimientos. Finalmente, retorna la lista de jugadas posibles para completar las perlas blancas.

El algoritmo generateWhiteCircleMoves[10]busca generar movimientos válidos para las perlas blancas que tienen exactamente dos vecinos. Itera sobre todas las celdas del tablero, identificando las perlas blancas con dos vecinos y luego llama a la función generateMovesFromWhiteCircle(node) para determinar los movimientos posibles para cada perla blanca. Luego, filtra las jugadas válidas utilizando la función checkJugadaWhite(node, posiblesJugadas). Si solo

**Algorithm 9** CompleteWhiteCircle()

```
1: procedure CompleteWhiteCircle
2:     jugadasPosiblesWhite ← []
3:     for row in 0 to gridSize − 1 do
4:         for col in 0 to gridSize − 1 do
5:             node ← getNode(row, col)
6:             if node and node.circleType = 1 and num_vecinos(node) = 1
   then
7:                 jugada ← completeLineThrough(node)
8:                 nodesource ← jugada.primero
9:                 nodedest ← jugada.segundo
10:                node.lineThrough ← true
11:                jugadasPosiblesWhite.append(jugada)
12:            end if
13:        end for
14:    end for
15:    return jugadasPosiblesWhite
16: end procedure
```

**Algorithm 10** generateWhiteCircleMoves()

```
1: procedure generateWhiteCircleMoves
2:     jugadasPosiblesWhite ← []
3:     jugadasHechas ← []
4:     for row in 0 to gridSize − 1 do
5:         for col in 0 to gridSize − 1 do
6:             node ← getNode(row, col)
7:             if node and node.circleType = 1 and num_vecinos(node) = 2
   then
8:                 posiblesJugadas ← generateMovesFromWhiteCircle(node)
9:                 filtroJugadas ← checkJugadaWhite(node, posiblesJugadas)
10:                if |filtroJugadas| = 1 then
11:                    jugar ← filtroJugadas.pop()
12:                    nodesource ← jugar.primero
13:                    nodedest ← jugar.segundo
14:                    jugadasHechas.append(jugar)
15:                end if
16:            end if
17:        end for
18:    end for
19:    return jugadasHechas
20: end procedure
```

hay una jugada válida, la realiza y la agrega a la lista de jugadas realizadas jugadasHechas. Finalmente, retorna la lista de jugadas realizadas.

El algoritmo generateMovesFromWhiteCircle [11] busca generar los movimientos posibles para una perla blanca dada. Comienza verificando si la perla blanca tiene una línea que atraviesa su ubicación utilizando la función verifyWhite(node). Si no hay una línea que atraviesa la perla blanca, el algoritmo continúa buscando las posibles jugadas para conectarla con sus vecinos. Dependiendo de la disposición de los vecinos, determina las direcciones en las que se pueden realizar movimientos adicionales para completar la línea blanca.

## 4.6 Complejidad

La complejidad de este algoritmo es$O(n^2)$, donde$n$ es el tamaño del tablero, ya que recorre todas las celdas del tablero en un bucle doble. Dentro de cada iteración del bucle, se realiza una verificación para determinar si la perla blanca tiene una línea a través de ella, lo que tiene una complejidad de tiempo constante$O(1)$. Luego, se generan posibles movimientos para conectar la perla blanca con sus vecinos, lo que también implica operaciones de tiempo constante para acceder a los nodos vecinos y realizar operaciones de lista. Por lo tanto, en general, la complejidad del algoritmo es dominada por el bucle principal, resultando en una complejidad total de$O(n^2)$.

## 4.7 Algoritmos para calcular jugadas espacios

El algoritmo GenerateBlankSpaceMove [12] busca generar movimientos posibles para una celda vacía en el tablero del juego. Itera sobre todas las celdas del tablero y verifica si la celda actual es una celda vacía (0) con exactamente un vecino. Si se cumple esta condición, busca los vecinos de la celda actual y evalúa si es posible realizar un movimiento desde la celda actual hacia uno de sus vecinos. Para considerar un movimiento como posible, el vecino no debe ser una celda vacía o una celda marcada como "deadSpot", y además, no debe haber una conexión previamente establecida entre la celda actual y el vecino. Si se encuentra un único movimiento posible, se realiza la jugada, marcando la celda actual como atravesada por una línea (lineThrough) y se registra la jugada realizada en una lista de jugadas hechas (jugadasHechas). Finalmente, el algoritmo devuelve la lista de jugadas hechas.

## 4.8 Complejidad

La complejidad de este algoritmo generateBlanckSpaceMove() es$O(N^2)$, donde$N$ es el tamaño del tablero. Esto se debe a que el algoritmo itera sobre todas las celdas del tablero (cada celda se recorre una vez) y realiza operaciones de tiempo constante en cada iteración. Por lo tanto, el tiempo de ejecución del algoritmo es proporcional al número total de celdas en el tablero, lo que resulta en una complejidad cuadrática.

**Algorithm 11** generateMovesFromWhiteCircle(node)

```
1: procedure GENERATEMOVESFROMWHITECIRCLE(node)
2:     posiblesJugadas ← []
3:     if ¬verifyWhite(node) then
4:         if node.vecinoabajo ≠ null and node.vecinoarriba ≠ null then
5:             nodeArriba ← node.vecinoarriba
6:             nodeAbajo ← node.vecinoabajo
7:             ArribaDerecha ← getNode(nodeArriba.row, nodeArriba.col + 1)
8:             ArribaIzquierda ← getNode(nodeArriba.row, nodeArriba.col−1)
9:             if ArribaDerecha ≠ null then
10:                posiblesJugadas.append({primero: nodeArriba, segundo: ArribaDerecha})
11:            end if
12:            if ArribaIzquierda ≠ null then
13:                posiblesJugadas.append({primero: nodeArriba, segundo: ArribaIzquierda})
14:            end if
15:            AbajoDerecha ← getNode(nodeAbajo.row, nodeAbajo.col + 1)
16:            AbajoIzquierda ← getNode(nodeAbajo.row, nodeAbajo.col − 1)
17:            if AbajoDerecha ≠ null then
18:                posiblesJugadas.append({primero: nodeAbajo, segundo: AbajoDerecha})
19:            end if
20:            if AbajoIzquierda ≠ null then
21:                posiblesJugadas.append({primero: nodeAbajo, segundo: AbajoIzquierda})
22:            end if
23:        else if node.vecinoderecha ≠ null and node.vecinoizquierda ≠ null
   then
24:            NodeDerecha ← node.vecinoderecha
25:            NodeIzquierda ← node.vecinoizquierda
26:            DerechaArriba ← getNode(NodeDerecha.row − 1, NodeDerecha.col)
27:            DerechaAbajo ← getNode(NodeDerecha.row + 1, NodeDerecha.col)
28:            if DerechaArriba ≠ null then
29:                posiblesJugadas.append({primero: NodeDerecha, segundo: DerechaArriba})
30:            end if
31:            if DerechaAbajo ≠ null then
32:                posiblesJugadas.append({primero: NodeDerecha, segundo: DerechaAbajo})
33:            end if
34:            IzquierdaArriba ← getNode(NodeIzquierda.row − 1, NodeIzquierda.col)
35:            IzquierdaAbajo ← getNode(NodeIzquierda.row + 1, NodeIzquierda.col)
36:            if IzquierdaArriba ≠ null then
37:                posiblesJugadas.append({primero: NodeIzquierda, segundo: IzquierdaArriba})
38:            end if
39:            if IzquierdaAbajo ≠ null then
40:                posiblesJugadas.append({primero: NodeIzquierda, segundo: IzquierdaAbajo})
41:            end if
42:        end if
43:    end if
44:    return posiblesJugadas
45: end procedure
```

**Algorithm 12** generateBlankSpaceMove()

```
1: procedure GENERATEBLANCKSPACEMOVE
2:     jugadasHechas ← []
3:     for row in 0 to gridSize − 1 do
4:         for col in 0 to gridSize − 1 do
5:             node ← getNode(row, col)
6:             jugadasPosiblesBlank ← []
7:             if node and node.circleType = null and num_vecinos(node) = 1
   then
8:                 vecino ← getAdjacentsall(row, col)
9:                 for veci in vecino do
10:                    if veci = null or veci.deadSpot then
11:                        continue
12:                    end if
13:                    if getAdjacents(veci.row, veci.col).length ≠ 2 and not
   isConnectionMade(row, col, veci.row, veci.col) then
14:                        jugadasPosiblesBlank.push({primero: node, segundo: veci})
15:                    end if
16:                end for
17:                if jugadasPosiblesBlank.length ≠ 1 then
18:                    continue
19:                end if
20:                jugada ← jugadasPosiblesBlank.pop()
21:                nodesource ← jugada.primero
22:                nodedest ← jugada.segundo
23:                jugadasHechas.push(jugada)
24:                node.lineThrough ← true
25:            end if
26:        end for
27:    end for
28:    return jugadasHechas
29: end procedure
```

## 4.9 Algoritmos de verificación

**Algorithm 13** checkJugada(jugadaDisponible)

```
1: procedure CHECKJUGADAWHITE(node, jugadaDisponible)
2:     findDeadSpots()
3:     verificate_play ← []
4:     for jugada in jugadaDisponible do
5:         nodeSource ← jugada.primero
6:         nodeDest ← jugada.segundo
7:         conec1 ← false
8:         if !isConnectionMade(nodeSource.row, nodeSource.col, nodeDest.row, nodeDest.col)
   then
9:             connectNodesByIndices(nodeSource.row, nodeSource.col, nodeDest.row, nodeDest.col)
10:            conec1 ← true
11:        end if
12:        if ifBranch(nodeSource) or ifBranch(nodeDest) or
   nodeSource.deadSpot == true or nodeDest.deadSpot == true or
   verifyInvalidLoop(node) then
13:            if conec1 == true then
14:                deleteConnectionByIndices(nodeSource.row, nodeSource.col, nodeDest.row, nodeDest.col)
15:            end if
16:        else
17:            verificate_play.push(jugada)
18:            if conec1 == true then
19:                deleteConnectionByIndices(nodeSource.row, nodeSource.col, nodeDest.row, nodeDest.col)
20:            end if
21:        end if
22:    end for
23:    return verificate_play
24: end procedure
```

El algoritmo checkJugada [13] se encarga de verificar si una jugada para una perla es válida, considerando varios criterios como la presencia de nodos de bifurcación, puntos muertos, la existencia de un bucle inválido y si la conexión entre los nodos de la jugada ya está realizada. Retorna una lista de jugadas verificadas que son válidas para realizar.

La función ifBranch [14] verifica si un nodo es un nodo de bifurcación, es decir, si tiene al menos tres vecinos. Retorna true si es un nodo de bifurcación y false en caso contrario.

El algoritmo isDeadSpot [15] determina si una celda específica del tablero es un "punto muerto". Un punto muerto es una celda vacía que no tiene vecinos y está bloqueada en al menos tres direcciones adyacentes. El algoritmo comprueba si la celda está vacía y sin vecinos, y luego verifica si al menos tres direcciones adyacentes están bloqueadas, donde una dirección se considera bloqueada si la celda vecina correspondiente tiene dos vecinos o es un punto muerto.

**Algorithm 14** ifBranch(node)

```
1: procedure IFBRANCH(node)
2:     if num_vecinos(node) ≥ 3 then
3:         return true
4:     end if
5:     return false
6: end procedure
```

**Algorithm 15** isDeadSpot(row, col)

```
1: procedure ISDEADSPOT(row, col)
2:     node ← getNode(row, col)
3:     if ¬node or node.circleType ≠ null then
4:         return false
5:     end if
6:     if num_vecinos(node) ≠ 0 then
7:         return false
8:     end if
9:     if node.deadSpot then
10:        return true
11:    end if
12:    blockedDirections ← 0
13:    adjacents ← getAdjacentsall(row, col)
14:    for dir in {3, 2, 1, 0} do                    ▷ Check each direction
15:        neighbor ← getNode(row + dir.row, col + dir.col)
16:        if ¬neighbor or num_vecinos(neighbor) = 2 or neighbor.deadSpot
   then
17:            blockedDirections ← blockedDirections + 1
18:        end if
19:    end for
20:    return blockedDirections ≥ 3
21: end procedure
```

**Algorithm 16** isConnectionMade($row1, col1, row2, col2$)

```
1: procedure ISCONNECTIONMADE(row1, col1, row2, col2)
2:     node1 ← getNode(row1, col1)
3:     node2 ← getNode(row2, col2)
4:     if node1 and node2 then
5:         return (node1.vecinoarriba = node2 or node1.vecinoabajo = node2
   or node1.vecinoderecha = node2 or node1.vecinoizquierda = node2)
6:     else
7:         return false
8:     end if
9: end procedure
```

El algoritmo isConnectionMade[16] verifica si existe una conexión entre dos nodos dados en el tablero del juego. Primero, obtiene los nodos correspondientes a las coordenadas proporcionadas. Luego, comprueba si ambos nodos existen y si alguno de los nodos vecinos de node1 coincide con node2. Si hay una conexión entre los dos nodos, devuelve verdadero; de lo contrario, devuelve falso.

**Algorithm 17** verifyInvalidLoop(*startNode*)

```
1:  procedure VERIFYINVALIDLOOP(startNode)
2:      visited ← set vacío
3:      stack ← [{node : startNode, parent : null}]
4:      perl ← 0
5:      while |stack|. > 0 do
6:          element ← stack.pop()
7:          node ← element.node
8:          parent ← element.parent
9:          if ¬visited.contains(node) then
10:             visited.add(node)
11:             if node.circleType = 1 then
12:                 perl ← perl + 1
13:             end if
14:             if node.circleType = 2 then
15:                 perl ← perl + 1
16:             end if
17:             neighbors ← [node.vecinoarriba, node.vecinoabajo, node.vecinoderecha, node.vecinoizquierda]
18:             for neighbor in neighbors do
19:                 if neighbor then
20:                     if ¬visited.contains(neighbor) then
21:                         stack.push({node : neighbor, parent : node})
22:                     else if neighbor ≠ parent then
23:                         if perl ≠ this.perls then
24:                             return true
25:                         end if
26:                     end if
27:                 end if
28:             end for
29:         end if
30:     end while
31:     return false
32: end procedure
```

El algoritmo verifyInvalidLoop [17] verifica si hay un ciclo inválido en el grafo del tablero del juego. Utiliza una estructura de pila para realizar una búsqueda en profundidad (DFS) desde el nodo de inicio dado. Durante la búsqueda, realiza un seguimiento de los nodos visitados y cuenta el número de perlas encontradas. Si encuentra un ciclo inválido, es decir, un nodo que ya ha sido visitado pero no

es el nodo padre inmediato, y el número de perlas en el ciclo no es igual al número total de perlas en el tablero, indica un ciclo inválido y devuelve verdadero. Si no se encuentra ningún ciclo inválido, devuelve falso.

## 4.10 Complejidad

La complejidad de la sección de verificación está dominada por el algoritmo verifyInvalidLoop, ya que realiza un recorrido en profundidad (DFS) en el grafo. La complejidad de este algoritmo puede variar considerablemente dependiendo de la estructura del tablero y las jugadas que se le pasen a la función. Sin embargo, en la mayoría de los casos, el número máximo de jugadas que se consideran es de 4, lo que hace que la complejidad sea \(O(1)\) en promedio para esta sección. Por otro lado, el algoritmo verifyInvalidLoop puede ser el de mayor complejidad, ya que realiza un recorrido DFS completo en el grafo del tablero, lo que implica una complejidad de tiempo $O(N)$, donde$N$ es el número de nodos en el grafo.

## 4.11 Algoritmos busqueda por eliminación

**Algorithm 18** GetIsland()

```
1:  procedure GetIsland
2:      Islas ← []
3:      for each nod in nodes do
4:          if num_vecinos(nod) = 0 and (nod.circleType = 1 or nod.circleType = 2) then
5:              validNeighbors ← getAllValidNeighbors(nod)
6:              print(nod, validNeighbors)
7:              Islas.push({node : nod, validNeighbors : validNeighbors})
8:          end if
9:      end for
10:     Islas.sort((a, b) → a.validNeighbors − b.validNeighbors)
11:     print(Islas)
12:     if Islas[0] then
13:         return Islas[0].node
14:     else
15:         return null
16:     end if
17: end procedure
```

El algoritmo GetIsland[18] busca identificar las perlas en un conjunto de nodos, donde una perla se define como un nodo que no tiene conexiones con otros nodos y que tiene un tipo de círculo específico (ya sea tipo 1 o tipo 2).

El proceso comienza inicializando una lista vacía llamada "Islas", que eventualmente contendrá las perlas encontradas.

Luego, se recorre cada nodo en el conjunto de nodos proporcionado. Para cada nodo, se verifica si tiene cero vecinos y si su tipo de círculo es 1 o 2. Si estas condiciones se cumplen, se obtienen todos los vecinos válidos del nodo utilizando la función "getAllValidNeighbors".

**Algorithm 19** conjeturaHorizontalBlanco(node)

```
1:  procedure conjeturaHorizontalBlanco(node)
2:      if node.circleType ≠ 1 then
3:          return null
4:      end if
5:      nodo_derecha ← getNode(node.row, node.col + 1)
6:      nodo_izq ← getNode(node.row, node.col − 1)
7:      posiblesJugadas ← []
8:      posiblesJugadaHorizontal ← null
9:      posiblesJugadas.push({primero : node, segundo : nodo_derecha})
10:     posiblesJugadas.push({primero : node, segundo : nodo_izq})
11:     posiblesJugadaHorizontal ← checkJugadaWhite(node, posiblesJugadas)
12:     if posiblesJugadaHorizontal.length = 2 then
13:         connectNodesByIndices(node.row, node.col, nodo_derecha.row, nodo_derecha.col)
14:         connectNodesByIndices(node.row, node.col, nodo_izq.row, nodo_izq.col)
15:         return posiblesJugadaHorizontal
16:     else
17:         return null
18:     end if
19: end procedure
```

El algoritmo conjeturaHorizontalBlanco [19] se encarga de verificar si un nodo dado tiene la posibilidad de responder con una jugada válida de tipo horizontal en un contexto específico. La condición inicial verifica si el tipo de círculo del nodo es igual a 1, ya que esta conjetura solo se aplica a nodos de este tipo.

Luego, se obtienen los nodos adyacentes a la derecha e izquierda del nodo dado. Se construye una lista de posibles jugadas con estos nodos. Esta lista se pasa a una función checkJugadaWhite para determinar si la jugada horizontal es válida o no. Si la función devuelve exactamente dos posibles jugadas, significa que la conjetura es válida.

El algoritmo conjetura_arriba_arriba_negro [20] se encarga de verificar si un nodo dado, que representa una perla negra, tiene la posibilidad de realizar una jugada válida de tipo vertical extendida en dos casillas hacia arriba.

Primero, se comprueba si el tipo de círculo del nodo es igual a 2, ya que esta conjetura solo se aplica a nodos de este tipo.

Luego, se obtienen los nodos adyacentes hacia arriba del nodo dado, es decir, el nodo directamente arriba (nodo_arriba) y el nodo dos casillas arriba (nodo_arriba_arriba). Si alguno de estos nodos es nulo (porque el nodo dado está en el borde superior del tablero), la función devuelve null, indicando que la

**Algorithm 20** conjetura_arriba_arriba_negro(node)

```
1: procedure CONJETURA_ARRIBA_ARRIBA_NEGRO(node)
2:     if node.circleType ≠ 2 then
3:         return null
4:     end if
5:     nodo_arriba ← getNode(node.row − 1, node.col)
6:     nodo_arriba_arriba ← getNode(node.row − 2, node.col)
7:     if nodo_arriba = null or nodo_arriba_arriba = null then
8:         return null
9:     end if
10:    posiblesJugadas ← []
11:    posiblesJugadaVertical ← null
12:    posiblesJugadas.push({primero : node, segundo : nodo_arriba})
13:    posiblesJugadas.push({primero : nodo_arriba, segundo :
   nodo_arriba_arriba})
14:    posiblesJugadaVertical = checkJugadaBlack(node, posiblesJugadas)
15:    if posiblesJugadaVertical.length = 2 then
16:        connectNodesByIndices(node.row, node.col, nodo_arriba.row, nodo_arriba.col)
17:        connectNodesByIndices(nodo_arriba.row, nodo_arriba.col, nodo_arriba_arriba.row, nodo_arriba
18:        return posiblesJugadaVertical
19:    else
20:        return null
21:    end if
22: end procedure
```

conjetura no es válida.

Después, se construye una lista de posibles jugadas con estos dos nodos. Esta lista se pasa a una función checkJugadaBlack para determinar si la jugada vertical extendida es válida o no. Si la función devuelve exactamente dos posibles jugadas, significa que la conjetura es válida.

### 4.12 Complejidad

Dado que la complejidad de los algoritmos checkJugadaBlack y checkJugadaWhite es $O(n)$, donde$n$ es el número de posibles jugadas a verificar, y considerando que $n$ es relativamente pequeño en comparación con el tamaño total del problema, podemos afirmar que la complejidad de las funciones conjetura_arriba_arriba_negro y conjeturaHorizontalBlanco es$O(1)$ con un componente $O(n)$ proveniente de estas funciones.

### 4.13 Complejidad total

Después de realizar un análisis exhaustivo de la complejidad de todos los algoritmos propuestos, se observa que la complejidad total es cuadrática, es decir, $O(n^2)$. Este resultado se obtuvo al determinar la complejidad de cada algoritmo por separado y encontrar que el factor dominante en términos de tiempo de ejecución es proporcional al cuadrado del tamaño del tablero, lo que lleva a una complejidad total cuadrática.

# Parte III
# Conclusión

Este proyecto ha sido revelador en cuanto a la complejidad inherente de resolver un problema SAT como el juego de Masyu. Descubrimos que abordar este desafío puede ser tanto una tarea compleja como extensa debido a la naturaleza combinatoria del problema y a la necesidad de considerar múltiples restricciones y condiciones. Sin embargo, a pesar de esta complejidad, también encontramos que existen diversas estrategias para enfrentar el problema.

En este proyecto, nos enfocamos en la aplicación de heurísticas para guiar las decisiones del jugador automático. Estas heurísticas se diseñaron para identificar patrones, reconocer jugadas clave y tomar decisiones estratégicas que maximicen las probabilidades de éxito en la resolución del juego. A través de algoritmos eficientes y métodos de optimización, buscamos mejorar continuamente el rendimiento y la efectividad del jugador automático.

Al implementar estas heurísticas, pudimos proporcionar al jugador automático la capacidad de tomar decisiones informadas y adaptativas en tiempo real, lo que mejoró significativamente su capacidad para resolver el juego de manera eficiente. Además, exploramos diferentes enfoques y técnicas para abordar los

desafíos específicos del juego, lo que nos permitió ampliar nuestro conocimiento y comprensión del problema.

Aunque reconocemos que la complejidad del problema SAT sigue siendo un desafío considerable, este proyecto nos ha brindado una valiosa experiencia en el desarrollo de soluciones prácticas y efectivas. Continuaremos explorando nuevas ideas y enfoques para mejorar aún más la capacidad del jugador automático y abordar otros problemas SAT con mayor eficacia en el futuro. En resumen, este proyecto no solo ha sido un ejercicio en la resolución de un problema complejo, sino también una oportunidad para aprender y crecer como desarrolladores y solucionadores de problemas.