Trabajo Práctico 3: Resolviendo un Nurikabe

Agustín Arias
Juan Zuccotti
Carlos Fernández

July 13, 2020

1 Introducción

1.1 Estructuras de Representación

a) Representamos cada grilla como una lista de listas. Por cuestiones de complejidad computacional de los algoritmos, también incluimos otros atributos que debe poseer una grilla.

$$
\begin{aligned}
Grilla == \langle & \\
& grilla : LISTA(LISTA(char)), \\
& total\_puntos : \mathbb{Z}, \\
& cantidad\_de\_numerales : \mathbb{Z}, \\
& hay\_cuadrado\_booleano : \mathbb{B}, \\
& se\_encontro\_sol : \mathbb{B} \\
\rangle &
\end{aligned}
$$

1.2 Invariantes de Representación

b) Para que una lista de listas represente correctamente una grilla debe ocurrir que:

- Todos los elementos de la lista de listas deben ser listas de la misma longitud.
- Los caracteres que aparecen en cada una de estas listas deben estar dentro de los permitidos. Es decir, cada uno de ellos debe ser un dígito entre 1 y 9, un punto o un numeral.
- total_puntos debe cumplir que tiene que ser un entero mayor o igual a cero y menor a la cantidad total de celdas. Además debe ser igual a la cantidad de puntos (celdas blancas) que debe tener una solución válida que se consiga a partir de la grilla.
- cantidad_de_numerales: debe cumplir que tiene que ser un entero mayor o igual a cero y menor a la cantidad total de celdas. Debe contar la cantidad de numerales que tiene la grilla.
- hay_cuadrado_booleano: Debe ser verdadero si y solo si existe una región cuadrada de 2×2 adentro de la grilla formada por paredes.
- se_encontro_sol: Debe ser verdadero si y solo si la grilla es una solución del nurikabe.

1.3 Algoritmos

c)

1. **Grilla**: Grilla(nombre_archivo)→Grilla:
   - Descripción:Construye una nueva grilla, leyéndola de un archivo de texto. La grilla solo puede contener números naturales entre el 1 y el 9, el punto . (celda en blanco) y numeral # (pared).
   - Precondición: El archivo forma una grilla rectangular que no tiene símbolos más allá de los mencionados arriba.

- Algoritmo: Se lee del archivo de entrada la grilla, poniéndola en una lista de listas en la cual cada elemento es uno de los siguientes: '0',..., '9', '.' o '#' y retorna la nueva instancia de Grilla.
- Complejidad $\mathbf{O}(m \times n)$: Se recorre una por una cada entrada del archivo y se la copia a una grilla. Como hay n renglones de m caracteres, la complejidad es la dada.

2. **g.es_posicion_valida**: g.es_posicion_valida(pos)$\rightarrow \mathbb{B}$:
   - Descripción: Dice si la posición *pos* de la grilla g está definida en la grilla.
   - Precondición: True.
   - Algoritmo:
     $$RV \leftarrow (pos[0] < g.ancho() \wedge pos[1] < g.alto() \wedge pos[0] \geq 0 \wedge pos[1] \geq 0)$$
   - Complejidad $\mathbf{O}(1)$: Consta únicamente de una asignación, y de algunas lecturas y comparaciones de variables.
3. **g.es_numero**: g.es_numero(pos)$\rightarrow \mathbb{B}$:
   - Descripción: Dice si la posición *pos* de la grilla g corresponde a un número.
   - Precondición: g.es_posicion_valida(*pos*).
   - Algoritmo: $RV \leftarrow g.grilla[pos[1]][pos[0]] \in$ "123456789"
   - Complejidad $\mathbf{O}(1)$: Es una asignación, acceso a variables y comparaciones. Todas realizadas en O(1).
4. **g.es_pared**: g.es_pared(pos)$\rightarrow \mathbb{B}$:
   - Descripción: Dice si la posición *pos* de la grilla g corresponde a una pared.
   - Precondición: g.es_posicion_valida(*pos*).
   - Algoritmo: $RV \leftarrow g.grilla[pos[1]][pos[0]] =' '$
   - Complejidad $\mathbf{O}(1)$: Es una asignación y solo hay comparaciones y accesos a posiciones.
5. **g.valor**: g.valor(*pos*)$\rightarrow \mathbb{N}$:
   - Devuelve el valor numérico de la posición *pos* en la grilla g.
   - Precondición: g.es_numero(*pos*).
   - Algoritmo: $RV \leftarrow int(g.grilla[pos[1]][pos[0]])$
   - Complejidad $\mathbf{O}(1)$: Es una asignación y solo hay accesos a variables y posiciones de una lista.
6. **g.alto**: g.alto()$\rightarrow \mathbb{N}$:
   - Devuelve el alto de la grilla g.
   - Precondición: True.
   - Algoritmo: $RV \leftarrow len(g.grilla)$
   - Complejidad $\mathbf{O}(1)$: Es una asignación.
7. **g.ancho**: g.ancho()$\rightarrow \mathbb{N}$:
   - Devuelve el ancho de la grilla g.
   - Precondición: True.
   - Algoritmo:

     $RV \leftarrow len(g.grilla[0])$
   - Complejidad $\mathbf{O}(1)$: Es una asignación.
8. **g.cantidad_no_paredes**: g.cantidad_no_paredes()$\rightarrow \mathbb{N}$:
   - Devuelve la cantidad de celdas de la grilla g que no corresponden a una celda de tipo pared.

- Precondición:True.
- Algoritmo:

  $RV \leftarrow g.alto() * g.ancho() - g.cantidad\_de\_numerales$
- Complejidad $\mathbf{O}(1)$: Es una resta, y asignación.

9. **g.hay_cuadrado**: g.hay_cuadrado()$\rightarrow \mathbb{B}$:
   - Determina si en g hay un cuadrado de 2×2 de paredes.
   - Precondición: True.
   - Algoritmo: Se recorre la lista de izquierda a derecha y de arriba a abajo buscando que haya un cuadrado de 2x2, formado por esa posición, la que esta a la derecha, la que esta abajo y la que esta en diagonal debajo de la que esta a la derecha.
   - Complejidad $\mathbf{O}(1)$: En la implementación este algoritmo lo hacemos en una función auxiliar en la cual cargamos el valor booleano en el atributo hay_cuadrado. En este caso, en la implementación esta función consiste solo de una asignación.
10. **g.islas_validas**: g.islas_validas()$\rightarrow \mathbb{B}$:
    - Determina si en la grilla g todos los números forman islas del tamaño que indican.
    - Precondición: True.
    - Algoritmo:

      for posición en posiciones que tienen números en sus celdas **do**
      while no se hayan recorrido todas las posiciones adyacentes a posición y que estén unidas por números o puntos. **do**
      if existe algún adyacente de pos que sea numero o punto **then**:
      OK
      else:
      salir y retornar False.
      end if
      end while
      end for
    - Complejidad $\mathbf{O}(m \times n)$: Como mucho, hay $m \times n$ posiciones con números y si todas ellas están una al lado de la otra, el algoritmo empezará con la primera posición y recorrerá toda la lista considerándolo como una sola parte conexa. En cualquier otro caso, por cada posición con números, solo tendrá que recorrer posiciones adyacentes a lo sumo ese número de veces, lo cual es trabajo constante (como mucho 9).
11. **g.pared_conexa**: g.pared_conexa()$\rightarrow \mathbb{B}$:
    - Determina si en la grilla g las paredes forman una única región conexa.
    - Precondición: True
    - Algoritmo:

      for posición en la lista de posiciones que son numerales **do**
      if existe algún adyacente de pos que esta en la lista mencionada **then**:
      OK
      else:
      salir y retornar False.
      end if
      end for

      Determina si en la grilla g, las paredes forman una única región conexa. Para eso se reutiliza la función es_conexa_via del trabajo práctico anterior con el caracter #.

- Complejidad $\mathbf{O}(m \times n)$: Este algoritmo como mucho se ejecutara $m \times n$ veces ya cada posición se visita a lo sumo una vez y en cada visita se miran las cuatro posiciones adyacentes, pero nunca se vuelven a visitar lugares por los que ya se pasó.

12. **g.resolver_nurikabe**: g.resolver_nurikabe(nombre_de_archivo)→ Grilla:

    - Descripción: Intenta resolver el nurikabe dado por g utilizando backtracking.
    - Precondición: g no contiene paredes.
    - Algoritmo: Para agilizar el algoritmo, se pintan previamente los adyacentes a todos los unos de las grilla, todas las celdas que tengan al menos dos adyacentes números y todas las islas con un solo punto.

      if se recorrió toda la grilla **then**:
      if se llego a una solución valida **then**:
      copiar la grilla a un archivo de salida
      end if
      Retornar la grilla (vacía si no se encontró una solución).
      end if
      if celda= blanco **then**
      Pinto celda de negro.
      if no hay cuadrado de paredes 2×2 y no hay ninguna isla con menos puntos de los requeridos **then**
      resolver(posicion_siguiente)
      end if
      Vuelvo a pintar celda de blanco.
      end if
      Llamar a resolver nurikabe con la siguiente posición