Introducción a la Computación

Primer Cuatrimestre 2019

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Informe

TP3

Integrante	LU	Correo electrónico
Bonardi Mabel	835/04	bonardimabel@gmail.com
Grevino Dante	225/13	dantegrevino@gmail.com
Robles David	576/06	roblesdavidr@gmail.com

1. Introducción

Un tipo de datos es una abstracción que determina un conjunto de valores que deben tener los datos de cierto tipo. Por ejemplo, en los lenguajes de programación se tienen el tipo Entero,Real y Booleano. A partir de esto es posible determinar que el 5 es un número entero y que, por ejemplo, true es un valor booleano, entre otros.

Además, los datos tienen ciertas operaciones y propiedades. Sabemos que para los números y los booleanos se tienen, respectivamente, las operaciones aritméticas y lógicas. Al considerar tanto datos como operaciones se tiene un tipo abstracto de datos.

Un tipo abstracto de datos (TAD) es un tipo de datos con sus operaciones. Se denomina abstracto porque nada en la definición indica el cómo se implementan las operaciones (puede haber más de una implementación).

2. Objetivo

En este trabajo se pretende crear una clase de estructura de datos llamada Mapa.

A partir de la estructura creada se implementará el tipo Mapa para representar un laberinto y, finalmente, mediante el uso de la *regla de la mano derecha*, saber si es posible dirigirse desde una posición inicial dada hasta una posición de destino.

3. Estructuras de Representación

En el momento de comenzar el diseño de un TAD es necesario tener una representación abstracta del objeto sobre el cual se requiere trabajar, sin necesidad de establecer un compromiso con ninguna estructura de datos concreta, ni con ningún tipo de lenguaje de programación seleccionado. Esto permite expresar las condiciones, relaciones, restricciones y operaciones de los elementos modelados, sin necesidad de restringirse a una representación interna concreta. Para ello, lo primero que se hace es dar nombre y estructura a los elementos a través de los cuales es posible modelar el estado interno de un objeto abstracto utilizando algún formalismo matemático o gráfico.

En nuestro caso particular, se utilizó únicamente el **TAD Mapa**. Según su diseño, su estructura de representación (que será privada) es:

$$Mapa == \langle matriz : LISTA,\ alto : \mathbb{Z},\ ancho : \mathbb{Z},\ paredes : \mathbb{Z} \rangle$$

donde $LISTA$ representa una lista de listas de enteros.

4. Invariantes de Representación

El invariante de un TAD establece una noción de validez para cada uno de sus objetos abstractos en términos de condiciones sobre su estructura interna y sus componentes. Esto es, indica en qué casos un objeto abstracto modela un elemento posible del mundo del problema. Por ejemplo, para un *TAD matriz* el invariante debe exigir que tanto el número de filas como el de columnas sean mayores a cero.

En particular, para el TAD Mapa, el invariante debe incluir condiciones como:

- La matriz es una lista donde cada elemento es otra lista y:
  (a) Una lista vacía es considerada una entrada válida.
  (b) La lista de la matriz está formada únicamente por listas de 0 o 1, donde los 0 representan los espacios vacíos y los 1 las paredes sólidas.
- Cada elemento de la lista debe tener la misma longitud. Esto se corresponde con el ancho de la matriz.
- La longitud de la lista se corresponde con el alto de la matriz.
- Alto, ancho y paredes deben ser enteros no-negativos.

Si un objeto del TAD no cumple cualquiera de estas condiciones, implica que no se encuentra modelando un elemento del TAD y por lo tanto la representación no es válida.

5. Análisis de Algoritmos

El análisis de la complejidad de algoritmos es una técnica que permite caracterizar la ejecución de algoritmos de manera independiente de cualquier plataforma, compilador o lenguaje.

En lugar de medir el tiempo de ejecución de un programa se mide el del algoritmo. Este estará programado en uno o más métodos. Así que la medida se basa en el examen del tiempo de ejecución para la invocación de un solo método. Al hacer más rápido un método se puede mejorar la velocidad de ejecución de un programa entero. Generalmente se calcula el tiempo de ejecución en el peor caso; de esta manera, se obtiene la mejor cota superior en el tiempo de ejecución para cualquier entrada posible.

Finalmente, en el análisis de la complejidad de un algoritmo no se incluyen constantes, lo que interesa al momento de comparar entre algoritmos es el orden de complejidad.

5.1. Algoritmo TAD Mapa

```
class Mapa:
  def __init__(self, nombre_archivo):
    if es_mapa(nombre_archivo):
      self.matriz = listar_texto(nombre_archivo)
      self.alto = len(self.matriz)
      self.paredes = 0
      if(self.alto > 0):
        self.ancho = len(self.matriz[0])
        fila = 0
        while fila <  self.alto:
          col = 0
          while col < self.ancho:
            self.paredes = self.paredes + self.matriz[fila][col]
            col = col +1
          fila = fila +1
      else:
        self.ancho = 0
```

A través del algoritmo se observa que la clase Mapa construye un nuevo mapa, leyéndolo de un archivo de texto en $O(longitud\ del\ archivo)$.

Por otra parte, aprovechando la complejidad disponible en la función constructor y, a medida que la matriz del mapa se va armando, se contabiliza la **cantidad de paredes**, el **alto** y **ancho** para guardarlo en una variable interna del tipo entero.

De esta manera, las operaciones **alto**, **ancho** y **cantidad de paredes** de un laberinto (que se definirá en la siguiente sección) serán $O(1)$.

5.2. Operaciones

- $m.es\_posicion\_valida(pos) \longrightarrow \mathbb{B}$: Dice si la posición pos está definida en el mapa m, en $O(1)$.
- $m.posicion(pos) \longrightarrow \mathbb{Z}$: Devuelve el valor de la posición pos del mapa m, en $O(1)$.

  Precondición: $m.es\_posicion\_valida(pos)$ debe ser $True$
- $m.alto() \longrightarrow \mathbb{Z}$: Devuelve el alto del mapa m, en $O(1)$.
- $m.ancho() \longrightarrow \mathbb{Z}$: Devuelve el ancho del mapa m, en $O(1)$.
- $m.cantidad\_paredes() \longrightarrow \mathbb{Z}$: Devuelve la cantidad de paredes mapa m, en $O(1)$.
- $m.corredor\_horizontal\_mas\_largo() \longrightarrow \mathbb{Z}$: Devuelve la longitud del corredor horizontal más largo del mapa m, en $O(ancho \times alto)$.
- $m.densidad\_arquitectonica() \longrightarrow \mathbb{Z}$: Devuelve la densidad arquitectónica del mapa m, en $O(1)$.
- $m.alcanzar\_con\_mano\_derecha(pos\_inicial, pos\_destino) \longrightarrow \mathbb{B}$: Devuelve $True$ si y solo si un explorador dentro de la grilla del mapa podría partir desde la posición inicial y llegar a la posición destino usando el método de la mano derecha.

  Se asume que el explorador recorre un espacio vacío por paso (respecto a la grilla: una posición para arriba, para abajo, para la izquierda, o para la derecha), y que no puede moverse en diagonal. La función debe devolver esta respuesta en $O(ancho \times alto)$.

  Precondición: $m.posicion(pos\_inicial) = 0$ (i.e. el explorador empieza en un espacio vacío). El mapa m no tiene bloques de espacios vacíos de $2x2$ o mayores.

5.3. Órdenes de Complejidad

Respecto a aquellos algoritmos que debían cumplir con un orden de complejidad $O(1)$ se observa que m.es_ posicion_ valida(pos), m.cantidad_ paredes(), m.alto() y m.ancho() verifican dicho orden ya que presentan instrucciones minimales que por definición son de orden constante.

En el caso de de m.posicion(pos), presenta una secuencialización. Sin embargo, cada una de las instrucciones también se corresponde con instrucciones minimales por lo que el orden máximo de dicha secuencialización es $O(1)$ y en consecuencia, el orden global será de $O(1)$. Finalmente, en el algoritmo de m.densidad_ arquitectonica() se presenta, además de la secuencialización de instrucciones con operaciones minimales que pertenecen a $O(1)$, un condicional. En ese caso, se evalúan el orden de la guarda, el del cuerpo del condicional y el del cuerpo del "else". A partir de ello, se toma como orden de complejidad del condicional al orden máximo hallado. En particular, dado que el condicional también se conforma por instrucciones minimales, es posible decir que el algoritmo analizado en su conjunto corresponde a $O(1)$.

Por otro lado, en lo que refiere a los algoritmos correspondientes a m.corredor_ horizontal_ mas_ largo() y m.alcanzar_ con_ mano_ derecha(pos_ inicial, pos_ destino), además de presentar secuencialización y condicionales con operaciones minimales, se presentan ciclos. Al momento de calcular el orden de complejidad de un ciclo, se calculan el orden de la guarda, el del cuerpo del ciclo y aquel orden que al que pertenecen la cantidad de iteraciones del ciclo. En el caso de m.corredor_ horizontal_ mas_ largo(), uno de los ciclos itera entre las filas hasta el alto del Mapa por lo que corresponde a $O(alto)$ y anidado, se encuentra otro ciclo, que itera entre las columnas hasta el ancho del Mapa, por lo que corresponde a $O(ancho)$. Al estar anidados, por definición se multiplican los órdenes de complejidad y como las demás instrucciones corresponden a $O(1)$, el orden de complejidad global del algoritmo será $O(alto \times ancho)$.

Finalmente, para m.alcanzar_ con_ mano_ derecha(pos_ inicial, pos_ destino), si bien no hay ciclos anidados, es posible llegar a $O(alto \times ancho)$ ya que al utilizar el método de la mano derecha si alguna vez se toca una posición en la que ya se estuvo, entonces esa posición es la nueva posición de partida. Eso prueba que a lo sumo se dan ancho x alto pasos. En consecuencia, el orden global pertenece a $O(alto \times ancho)$.

5.4. Algoritmo de las Operaciones en Pseudocódigo

Algorithm 1 | m.es_ posicion_ valida(pos):

Entrada: Un par de puntos de enteros ordenados.
Salida: Verifica si la posición se encuentra en el laberinto.
initialization
 $RV \longleftarrow$ True o False dependiendo de si la posicion existe o no en la matriz.

Algorithm 2 | m.posicion(pos):

Entrada: Un par de puntos de enteros ordenados.
Salida: El valor que hay en esa posición en el laberinto.
initialization
 if *(La posicion pedida es valida dentro de la matriz):* **then**
 | Posicion, se asigna el valor que tiene en ese lugar
end

Algorithm 3 | m.alto():

Entrada: Laberinto
Salida: El alto del laberinto
initialization
 if *(Si el Mapa es no vacío)* **then**
 | Alto, asigna a la cantidad de listas que tiene la matriz
else
 | Alto, asigna el valor cero
end

Algorithm 4 | m.ancho():

Entrada: Laberinto.
Salida: El ancho del laberinto.
initialization
if *(Si el Mapa es no vacío)* **then**
| Ancho, asigna a la longitud de la primer fila
else
| Ancho, asigna el valor cero
end

Algorithm 5 | cantidad_ paredes():

Entrada: Laberinto.
Salida: La cantidad de 1's; es decir, paredes que contiene el laberinto.
initialization
Paredes, asigna la cantidad de paredes al inicializar la clase de MAPA

Algorithm 6 | densidad_ arquitectonica():

Entrada: Laberinto.
Salida: Proporcion de paredes que hay en el laberinto.
initialization
Paredes, se asigna la cantidad de 1's que hay en el mapa
Total, se asignan las dimensiones del mapa
if *(Si es un Mapa vacío):* **then**
| Densidad del mapa será cero
else
| Densidad corresponde a la división entre la cantidad de paredes y el Total
end

Algorithm 7 | corredor_ horizontal_ mas_ largo():

Entrada: Laberinto.
Salida: Secuencia horizontal más larga de espacios vacíos.
initialization
mayor_longitud_de_ceros ⟵ 0
if *(Si el Mapa es no vacío)* **then**
| **while** *(Recorro las Filas)* **do**
| | temporal_longitud_de_ceros ⟵ 0
| | **while** *(Recorre las Columnas)* **do**
| | | **if** *(Encuentra un* 0 *en esa posición de la matriz)* **then**
| | | | temporal_longitud_de_ceros ⟵ mayor_longitud_de_ceros + 1
| | | | **if** *(Si el maximo temporal es mayor que el final)* **then**
| | | | | actualiza el valor del mayor temporal con el temporal
| | | **else**
| | | | reinicia la temporal_longitud_de_ceros desde cero
| | | **end**
| | **end**
| **end**
end
RV ⟵ *mayor_longitud_de_ceros*

Algorithm 8 | alcanzar_ con_ mano_ derecha (pos_ inicial, pos_ destino):

Entrada: Laberinto junto con una posición inicial y otra de destino.
Salida: TRUE si es posible hallar un camino entre la posición incial y la de destino, FALSE en caso contrario.

```
initialization
 pos_actual ⟵ pos_inicial
 respuesta ⟵ False
 if (Posicion Inicial es igual a la Posicion Destino) then
 | respuesta⟵ True /* caso trivial                                    */
end
/* empiezo hacia la derecha, si se puede                              */
if (A la derecha de mi posicion tengo un cero & Abajo de mi posicion tengo un 1): then
 | Posicion Anterior ⟵ Posicion Inicial
 |  Posicion Actual ⟵ Posicion a la derecha de la inicial
 |  while (Posicion Actual sea distinta de la inicial & respuesta == False): do
 |  | if Posicion Actual es igual a la Destino then
 |  |  | respuesta ⟵ True
 |  | else
 |  |  | Auxiliar⟵ Posicion Actual
 |  |  |  Posicion Actual⟵ avanzar(Posicion Anterior, Posicion Actual)
 |  |  |  Posicion Anterior⟵ Auxiliar
 |  | end
 | end
end
/* empiezo hacia arriba, si se puede                                  */
if (Arriba de mi posicion inicial tengo un cero & A la derecha de mi posicion inicial tengo un 1): then
 | Posicion Anterior ⟵ Posicion Inicial
 |  Posicion Actual ⟵ Posicion arriba de la inicial
 |  while (Posicion Actual sea distinta de la inicial & respuesta == False): do
 |  | if Posicion Actual es igual a la Destino then
 |  |  | respuesta ⟵ True
 |  | else
 |  |  | Auxiliar⟵ Posicion Actual
 |  |  |  Posicion Actual⟵ avanzar(Posicion Anterior, Posicion Actual)
 |  |  |  Posicion Anterior⟵ Auxiliar
 |  | end
 | end
end
/* empiezo hacia la izquierda, si se puede                            */
if (A la izquierda de mi posicion tengo un cero & Arriba de mi posicion inicial tengo un 1): then
 | Posicion Anterior ⟵ Posicion Inicial
 |  Posicion Actual ⟵ Posicion izquierda de la inicial
 |  while (Posicion Actual sea distinta de la inicial & respuesta == False): do
 |  | if Posicion Actual es igual a la Destino then
 |  |  | respuesta ⟵ True
 |  | else
 |  |  | Auxiliar⟵ Posicion Actual
 |  |  |  Posicion Actual⟵ avanzar(Posicion Anterior, Posicion Actual)
 |  |  |  Posicion Anterior⟵ Auxiliar
 |  | end
 | end
end
/* empiezo hacia abajo, si se puede                                   */
if (Abajo por la derecha de mi posicion inicial tengo un cero & A la izquierda de mi posicion inicial tengo un 1): then
 | Posicion Anterior ⟵ Posicion Inicial
 |  Posicion Actual ⟵ Posicion abajo de la inicial
 |  while (Posicion Actual sea distinta de la inicial & respuesta == False): do
 |  | if Posicion Actual es igual a la Destino then
 |  |  | respuesta ⟵ True
 |  | else
 |  |  | Auxiliar⟵ Posicion Actual
 |  |  |  Posicion Actual⟵ avanzar(Posicion Anterior, Posicion Actual)
 |  |  |  Posicion Anterior⟵ Auxiliar
 |  | end
 | end
end
```

6. Funciones Auxiliares

Por último, se definen las funciones auxiliares.

- *listar_texto(texto)* $\longrightarrow$ *LISTA*: Lee un archivo y lo convierte en una lista de las filas.

Algorithm 9 | listar_ texto(texto):

Entrada: Archivo de texto.
Salida: Lista de listas.
initialization
 lista_ de_ filas$\longleftarrow$ archivo de lectura abierto
while *(Recorre la Cantidad de Listas):* **do**
 | Enumera los elementos dentro de la lista
end

- *es_mapa(nombre_de_archivo)* $\longrightarrow \mathbb{B}$: Devuelve True o False dependiendo de si el texto input es un mapa o no.

Algorithm 10 | es_ mapa(texto):

Entrada: Archivo de texto.
Salida: Lista de listas.
initialization
 Archivo, asigna la matriz formada usando la funcion anterior
 RV $\longleftarrow$ True
 if *(El Mapa es no vació)* **then**
 | Ancho, asigna la longitud de la primer fila
 | **if** *(Si el ancho es cero)* **then**
 | | No se considera como Mapa
 | | **while** *(Recorre la cantidad de filas)* **do**
 | | | **if** *(EL largo de las filas es igual a la primera)* **then**
 | | | | **while** *(Recorre las columnas)* **do**
 | | | | | **if** *(Los elementos de matriz son distintos de* 0 *o* 1*)* **then**
 | | | | | | No se considera como Mapa
 | | | | | **end**
 | | | | **end**
 | | | **else**
 | | | | No se condiera como Mapa
 | | | **end**
 | | **end**
 | **end**
end

- Algoritmos de movimiento:

Algorithm 11 | derecha():

Entrada: Posición en el Mapa.
Salida: Posición a la derecha de la inicial.
initialization
RV $\longleftarrow$ moverme un lugar a la derecha

Algorithm 12 | izquierda():

Entrada: Posición en el Mapa.
Salida: Posición a la izquierda de la inicial.
initialization
RV $\longleftarrow$ moverme un lugar a la izquierda

Algorithm 13 | arriba():

Entrada: Posiciónen el Mapa.
Salida: Posición arriba de la inicial.
initialization
RV $\longleftarrow$ moverme un lugar hacia arriba

Algorithm 14 | abajo():

Entrada: Posiciónen el Mapa.
Salida: Posición abajo de la inicial.
initialization
$RV \longleftarrow$ moverme un lugar hacia abajo

Algorithm 15 | avanzar():

Entrada: Posicion en el Mapa
Salida: Posicion en el Mapa
initialization
if *(Posicion Actual esta a la Derecha de la Posicion Anterior)* **then**
  if *(Abajo de la Posicion Actual hay un Cero)* **then**
    Nueva Posicion, asigno la Posicion Abajo de la Actual
  else
    if *(A la Derecha de la Posicion Actual hay un Cero)* **then**
      Nueva Posicion, asigno la Posicion Derecha de la Actual
    else
      if *(Arriba de la Posicion Actual hay un Cero)* **then**
        Nueva Posicion, asigno la Posicion Arriba de la Actual
      else
        Nueva Posicion, asigno Posicion Anterior
      end
    end
  end
end

if *(Posicion Actual esta Arriba de la Posicion Anterior)* **then**
  if *(A la Derecha de la Posicion Actual hay un Cero)* **then**
    Nueva Posicion, asigno la Posicion a la Derecha a la Actual
  else
    if *(Arriba de la Posicion Actual hay un Cero)* **then**
      Nueva Posicion, asigno la Posicion Arriba de la Actual
    else
      if *(A la Izquierda de la Posicion Actual hay un Cero)* **then**
        Nueva Posicion, asigno la Posicion a la Izquierda de la Actual
      else
        Nueva Posicion, asigno Posicion Anterior
      end
    end
  end
end

if *(Posicion Actual esta a la Izquierda de la Posicion Anterior)* **then**
  if *(Arriba de la Posicion Actual hay un Cero)* **then**
    Nueva Posicion, asigno la Posicion Arriba de la Actual
  else
    if *(A la Izquierda de la Posicion Actual hay un Cero)* **then**
      Nueva Posicion, asigno la Posicion Izquierda de la Posicion Actual
    else
      if *(Abajo de la Posicion Actual hay un Cero)* **then**
        Nueva Posicion, asigno la Posicion Abajo de la Actual
      else
        Nueva Posicion, asigno Posicion Anterior
      end
    end
  end
end

if *(Posicion Actual esta Abajo de la Posicion Anterior)* **then**
  if *(A la Izquierda de la Posicion Actual hay un Cero)* **then**
    Nueva Posicion, asigno la Posicion Izquierda de la Actual
  else
    if *(Abajo de la Posicion Actual hay un Cero)* **then**
      Nueva Posicion, asigno la Posicion Abajo de la Actual
    else
      if *(A la Derecha de la Posicion Actual hay un Cero)* **then**
        Nueva Posicion, asigno la Posicion Derecha de la Actual
      else
        Nueva Posicion, asigno Posicion Anterior
      end
    end
  end
end