Algoritmos y Estructuras de Datos II

Trabajo Práctico 3

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

Pacalgo2

Los inertes

Integrante	LU	Correo electrónico
Bruno Robbio	480/09	brobbio@hotmail.com
Nicolas Andres Kinaschuk	248/15	nicolaskinaschuk@gmail.com
Pedro Joel Burgos	804/18	facultadburgospedrojoel@hotmail.com
Valentina Madelaine Saravia Ruiz	257/18	valentina.saraviaruiz@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Modulo Mapa

Interfaz

se explica con: MAPA

géneros: `mapa`.

Operaciones básicas de mapa

NUEVOMAPA(**in** *largo*: `nat`, **in** *alto*: `nat`, **in** *inicio*: `coordenada`, **in** *llegada*: `coordenada`, **in** *fantasmas*: `conj(coordenada)`, **in** *paredes*: `conj(coordenada)`, **in** *chocolates*: `conj(coordenada)`) → *res* : `mapa`
Pre ≡ {(*inicio* ≠ *llegada* ∧ *todosEnRango*(*paredes* ∪ *fantasmas* ∪ *chocolates* ∪ {*inicio*, *llegada*}, *largo*, *alto*) ∧ {*inicio*, *llegada*} ∩ (*fantasmas* ∪ *paredes*) = ∅ ∧ *disjuntosDeAPares*(*paredes*, *fantasmas*, *chocolates*))}
Post ≡ {*res* = *nuevoMapa*(*largo*, *alto*, *inicio*, *llegada*, *paredes*, *fantasmas*, *chocolates*))}
Complejidad: $O(alto \cdot largo \cdot (\#chocolates + \#fantasmas + \#paredes + 1))$
Descripción: Genera un nuevo mapa
Aliasing: Para construir el mapa hacemos copia de todos los conjuntos

ESCASILLEROPELIGROSO(**in** *m*: `mapa`, **in** *posicion*: `coordenada`) → *res* : `bool`
Pre ≡ {true}
Post ≡ { res = distConFantasmasMasCercano(fantasmas(m), posicion) ≤ 3 }
Complejidad: $O(1)$
Descripción: Devuelve true si el casillero es peligroso, es peligroso si existe un fantasma con distancia ≤ 3 respecto a la posción

ENRANGO(**in** *m*: `mapa`, **in** *posicion*: `coordenada`) → *res* : `bool`
Pre ≡ {true}
Post ≡ {*res* = enRango(posicion, largo(m), alto(m))}
Complejidad: $O(1)$
Descripción: Devuelve true si la posicion se encuentra en rango

CANTCHOCOLATES(**in** *map*: `mapa`) → *res* : `nat`
Pre ≡ {true}
Post ≡ {*res* = #(chocolates(map))}
Complejidad: $O(c)$
Descripción: Devuelve la cantidad de chocolates en el mapa

ESPARED(**in** *map*: `mapa`, **in** *posicion*: `coordenada`) → *res* : `bool`
Pre ≡ {true}
Post ≡ {*res* = true ⟺ *posicion* ∈ paredes(map)}
Complejidad: $O(1)$
Descripción: Devuelve el conjunto de paredes

INICIO(**in** *map*: `mapa`) → *res* : `coordenada`
Pre ≡ {true}
Post ≡ {*res* = inicio(map)}
Complejidad: $O(1)$
Descripción: Devuelve la coordenada de inicio del mapa

LLEGADA(**in** *map*: `mapa`) → *res* : `coordenada`
Pre ≡ {true}
Post ≡ {*res* = llegada(map)}
Complejidad: $O(1)$
Descripción: Devuelve la coordenada de llegada del mapa

IDCHOCOLATE(**in** m : `mapa`, **in** *posicion* : `coordenada`) $\rightarrow$ *res* : `int`
Pre $\equiv$ {enRango(*posicion*)}
Post $\equiv$ {*posicion* $\in$ chocolates(m) $\iff$ $0 \leq res <$ #chocolates(m)}
Complejidad: $O(1)$
Descripción: Devuelve el id del chocolate en el mapa

Implementación

Representación

mapa se representa con *mp*

donde *casillero* es tupla(*fantasma*: bool,
peligrosa: bool,
pared: bool,
idChocolate: int)

donde *columna* es array[0...*largo*] de *casillero*

donde *mp* es tupla(*matriz*: array[0...*alto*] de *columna*,
#chocolates: nat,
alto: nat,
largo: nat,
inicio: coordenada,
llegada: coordenada)

Invariante de representación

Rep : mp $\longrightarrow$ boolean

$$
\begin{aligned}
\text{Rep}(e) \equiv\ & \text{True} \iff (0 \leq e.inicio_1 < e.largo \wedge 0 \leq e.inicio_2 < e.alto\ ) \wedge \\
& (0 \leq e.llegada_1 < e.largo \wedge 0 \leq e.llegada_2 < e.alto\ ) \wedge \\
& (e.inicio \neq e.llegada) \wedge \\
& (\forall\ \text{i: nat})(0 \leq \text{i} < e.largo) \ \Rightarrow_L (\\
& (\forall\ \text{j: nat})(0 \leq \text{j} < e.alto) \ \Rightarrow_L (\\
& \quad (\beta(e.matriz[i][j].pared) + \\
& \quad\ \beta(e.matriz[i][j].fantasma) + \\
& \quad\ \beta(0 \leq e.matriz[i][j].idChocolate < e.\#chocolates) \leq 1) \wedge \\
& \quad ((\forall\ n\text{: } \mathbb{N})(0 \leq n < e.\#chocolates) \ \Rightarrow \\
& \quad\ (\exists!\ i,j\text{: } nat\ )(\ 0 \leq i < e.largo \wedge 0 \leq j < e.alto\ ) \wedge_L \\
& \quad\ (\ e.matriz[i][j].chocolates = n\ )) \wedge \\
& \quad (e.matriz[i][j].peligrosa \ \Rightarrow_L (\exists\ n,m\text{: nat})(0 \leq n < e.largo \wedge 0 \leq m < e.alto) \wedge_L \\
& \quad\ (e.matriz[n][m].fantasma \wedge \text{distancia}(\langle i,j\rangle, \langle n,m\rangle) \leq 3))) \\
&)
\end{aligned}
$$

Función de abstracción

Abs : mp e $\longrightarrow$ Mapa {Rep(e)}

$(\forall\ e : \text{mp})\ \text{Abs}(e) =_{\text{obs}} m$: mapa | largo($m$) = $e.largo \wedge$
alto(m) = $e.alto \wedge$
#(chocolates(m)) = $e.\#chocolates \wedge$
$e.inicio = inicio(m) \wedge$
$e.llegada = llegada(m) \wedge$
$(\forall$ i: nat$)(0 \leq \text{i} < e.largo) \Rightarrow_L ($
$(\forall$ j: nat$)(0 \leq \text{j} < e.alto) \Rightarrow_L ($
$e.matriz[i][j].fantasma \iff \langle i, j\rangle \in fantasmas(m) \wedge$
$e.matriz[i][j].paredes \iff \langle i, j\rangle \in paredes(m) \wedge$
$0 \leq e.matriz[i][j].idChocolate < e.\#chocolates \iff \langle i, j\rangle \in chocolates(m) \wedge$
$e.matriz[i][j].peligrosa \iff distConFantasmasMasCercano(fantasmas(m), \langle i, j\rangle) \leq 3))$

Algoritmos

---

iNuevoMapa(**in** *largo*: `nat`, **in** *alto*: `nat`, **in** *inicio*: `coordenada`, **in** *llegada*: `coordenada`, **in** *fantasmas*: `conj(coordenada)`, **in** *paredes*: `conj(coordenada)`, **in** *chocolates*: `conj(coordenada)`) $\rightarrow$ *res* : *mapa*

1: *int contadorDeChocolate* $\leftarrow 0$ ▷ $\mathcal{O}(1)$
2: *mapa* $\leftarrow <>$ ▷ $\mathcal{O}(1)$
3: *mapa.inicio* $\leftarrow$ *inicio* ▷ $\mathcal{O}(1)$
4: *mapa.llegada* $\leftarrow$ *llegada* ▷ $\mathcal{O}(1)$
5: *mapa.largo* $\leftarrow$ *largo* ▷ $\mathcal{O}(1)$
6: *mapa.alto* $\leftarrow$ *alto* ▷ $\mathcal{O}(1)$
7: *mapa.#chocolates* $\leftarrow$ *chocolates.longitud* ▷ $\mathcal{O}(1)$
8: **para** *int* $x = 0; x <$ *mapa.largo*$; x++$ **hacer** ▷ $\mathcal{O}(largo)$
9: **para** *int* $y = 0; y <$ *mapa.alto*$; y++$ **hacer** ▷ $\mathcal{O}(largo * alto)$
10: *mapa.matriz*$[x][y]$*.pared* $= (x, y)$ *in paredes* ▷ $\mathcal{O}(largo * alto * |P|)$
11: *mapa.matriz*$[x][y].fantasma = (x, y)$ *in fantasmas* ▷ $\mathcal{O}(largo * alto * |F|)$
12: *mapa.matriz*$[x][y]$*.peligorsa* $=$ *seAsusta*$((x, y), fantasmas)$ ▷ $\mathcal{O}(largo * alto * |F|)$
13: **si** (x, y) *in chocolates* **entonces** ▷ $\mathcal{O}(largo * alto * |C|)$
14: *mapa.matriz*$[x][y]$*.idChocolate* $\leftarrow$ *contadorDeChocolate* ▷ $\mathcal{O}(largo * alto * |C|)$
15: *contadorDeChocolate*$++$ ▷ $\mathcal{O}(largo * alto * |C|)$
16: **else**
17: *mapa.matriz*$[x][y]$*.idChocolate* $\leftarrow -1$ ▷ $\mathcal{O}(largo * alto * |C|)$
18: *res* $\leftarrow$ *mapa* ▷ $O(1)$

Complejdad: $\mathcal{O}((alto \cdot largo \cdot (\#chocolates + \#fantasmas + \#paredes + 1))$

---

seAsusta(**in** c: `coordenada`, **in** $fantasmas$: `conj(coordenada)` **out** res: `bool`)

```
1: res ← False
2: int i ← 0                                                          ▷ O(1)
3: mientras i < fantasma.longitud hacer                               ▷ O(|f|)
4:     si distancia(c, fantasma[i]) ≤ 3 entonces                      ▷ O(|f|)
5:         res ← true                                                 ▷ si es el peor caso no entra
6:     else
7:         i++                                                        ▷ O(|f|)
```

Complejidad: $\mathcal{O}(|f|)$

---

distancia(**in** c: `coordenada`, **in** f: `coordenada` **out** res: `nat`)

```
1: res ← valorAbsoluto(c_1 − f_1) + valorAbsoluto(c_2 − f_2)          ▷ Θ(1)
```

Complejidad: $\Theta(1)$

---

iesCasilleroPeligroso(**in** m: `mapa`, **in** $posicion$: `coordenada`) → res : $bool$

```
1: si enRango(m, posicion) ∧_L m.matriz.[posicion_1][posicion_2].peligrosa entonces   ▷ Θ(1)
2:     res ← true                                                     ▷ Θ(1)
3: else
4:     res ← false                                                    ▷ Θ(1)
```

Complejidad: $\Theta(1)$

---

ienRango(**in** m: `mapa`,**in** $posicion$: `coordenada`) → res : $bool$

```
1: res ← (0 ≤ posicion_1 < m.alto ∧_L 0 ≤ posicion_2 < m.largo)       ▷ Θ(1)
```

Complejidad: $\Theta(1)$

---

iCantChocolates(**in** map: `mapa`) → res : nat

```
1: res ← map.#chocolates                                              ▷ Θ(1)
```

Complejidad: $\Theta(1)$

---

iEsPared(**in** map: `mapa`, **in** $posicion$: `coordenada`) → res : $bool$

```
1: res ← map.matriz.[posicion_1][posicion_2].pared                    ▷ Θ(1)
```

Complejidad: $\Theta(1)$

---

iInicio(**in** map: `mapa`) → res : $coordenada$

```
1: res ← map.inicio                                                   ▷ Θ(1)
```

Complejidad: $\Theta(1)$

---

iLlegada(**in** map: `mapa`) → res : $coordenada$

```
1: res ← map.llegada                                                  ▷ Θ(1)
```

Complejidad: $\Theta(1)$

iIdChocolate(**in** m : `mapa`, **in** $posicion$: `coordenada`) $\rightarrow$ res : int

1: $res \leftarrow m.matriz.[posicion_1][posicion_2].idChocolate$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

Servicios usados

-

Modulo Partida

Interfaz

se explica con: PARTIDA

géneros: `partida`.

Operaciones básicas de partida

NUEVAPARTIDA(**in** m : `mapa`) $\rightarrow$ res : `partida`
Pre $\equiv$ {true}
Post $\equiv$ {res = nuevaPartida(m)}
Complejidad: $O(c)$, c es la cantidad de chocolates que contiene el mapa
Descripción: Genera una nueva partida
Aliasing: El mapa se recibe por referencia

MOVER(**in/out** p : `partida`, **in** d : `dirección`)
Pre $\equiv$ {$p_0 = p$}
Post $\equiv$ {p = mover(p_0, d)}
Complejidad: $O(1)$
Descripción: Mueva la posición del jugador un casillero
Aliasing: Se modifica p internamente

GANÓ?(**in** p : `partida`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res = ganó?(p)}
Complejidad: $O(1)$
Descripción: Devuelve true si el jugador gano la partida

PERDIÓ?(**in** p : `partida`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res = perdió?(p)}
Complejidad: $O(1)$
Descripción: Devuelve true si el jugador perdio la partida

JUGADOR(**in** p : `partida`) $\rightarrow$ res : `coordenada`
Pre $\equiv$ {true}
Post $\equiv$ {res = perdió?(p)}
Complejidad: $O(1)$
Descripción: Devuelve la posicion del jugador

CANTMOV(**in** p : `partida`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {res = cantMov(p)}
Complejidad: $O(1)$
Descripción: Devuelve la cantidad de movimientos del jugador

Implementación

Representación

partida se representa con *pt*

donde *pt* es tupla(*mapa*: mp,
jugador: coordenada,
chocolates: array[0...c] de bool,
cantMov: nat,
inmunidad: nat,
gano: bool,
perdio: bool)

Funciones auxiliares

distancia : coordenada × coordenada $\longrightarrow$ nat

distanciaMinima : coordenada × conj(coordenada) $\longrightarrow$ bool

chocolatesSinComer : pt $\longrightarrow$ conj(coordenada)

distancia$(x, y) \equiv |+x_1 \text{ - } +y_1| + |+x_2 \text{ - } +y_2|$

distanciaMinima$(j, c) \equiv$ **if** $\#(c) = 1$ **then**
distancia(j, dameUno(c))
else
mín(distancia(j, dameUno(c)),distanciaMinima(j, sinUno(c)))
fi

chocolatesSinComer(e) $\equiv$ Esta función devuelve el conjunto de coordenadas de chocolates en la $e.mapa.matriz$ cuyo Id en el array de la partida ($e.chocolates$) aun estan en true

Invariante de representación

Rep : pt $\longrightarrow$ boolean

Rep(e) $\equiv$ True $\iff$ ($mapa.EnRango(e.mapa, e.jugador) \wedge$
$long(e.chocolates) = e.mapa.\#chocolates) \wedge_L$
$e.cantMov = 0 \Rightarrow$
$e.mapa.inicio = jugador \wedge$
if $0 \leq e.mapa.matriz[jugador_1][jugador_2].idChocolate < e.mapa.\#chocolates$ **then**
$e.chocolates[e.mapa.matriz[e.jugador_1][e.jugador_2].idChocolate] = false \wedge$
$e.inmunidad = 10 \wedge$
$(\forall i : \mathbb{N})(0 \leq i < mapa.\#chocolates \wedge i \neq e.mapa.matriz[e.jugador_1][e.jugador_2].idChocolate)$
$\Rightarrow {}_L(e.chocolates[i] = true)$
else
$e.inmunidad = 0 \wedge (\forall i : \mathbb{N})(0 \leq i < mapa.\#chocolates) \Rightarrow {}_L(e.chocolates[i] = true)$
fi $\wedge$
$(e.chocolates[e.mapa.matriz[e.jugador_1][e.jugador_2].idChocolate] = false) \wedge$
$e.inmunidad$=10 $\Rightarrow 0 \leq e.mapa.matriz[e.jugador_1][e.jugador_2].idChocolate < e.mapa.\#chocolates \wedge$
$e.inmunidad \leq$ 10 - distanciaMinima($jugador$, chocolatesSinComer(e)) $\wedge$
$e.gano \iff jugador = e.mapa.llegada \wedge$
$e.perdio \iff e.inmunidad = 0 \wedge e.mapa.distConFantasmasMásCercano(e.mapa, e, jugador) \leq 3$

Función de abstracción

Abs : pt e $\longrightarrow$ partida {Rep(e)}

($\forall$ e : pt) Abs(e) $=_{\text{obs}}$ p: partida | mapa(p) = $e.mapa$ $\wedge$
jugador(p) = $e.jugador$ $\wedge$
chocolates(p) = chocolatesSinComer(e) $\wedge$
cantMov(p) = $e.cantMov$ $\wedge$
inmunidad(p) = $e.inmunidad$ $\wedge$
ganó?(p) = $e.gano$ $\wedge$
perdió?(p) = $e.perdio$

Algoritmos

---

iNuevaPartida(**in** m : `mapa`) $\rightarrow$ res : $partida$

1: $partida \leftarrow <>$ ▷ $\mathcal{O}(1)$
2: $partida.mapa \leftarrow m$ ▷ $\mathcal{O}(1)$
3: $partida.jugador \leftarrow mapa.inicio(m)$ ▷ $\mathcal{O}(1)$
4: $i \leftarrow 0$ ▷ $\mathcal{O}(1)$
5: **mientras** $i < mapa.cantChocolates(m)$ **hacer** ▷ $\mathcal{O}(c)$
6: $\quad partida.chocolates[i] \leftarrow true$ ▷ $\mathcal{O}(c)$
7: $\quad i++$ ▷ $\mathcal{O}(c)$
8: $partida.cantMov \leftarrow 0$ ▷ $\mathcal{O}(1)$
9: $idChocolate \leftarrow mapa.idChocolate(m, partida.jugador)$ ▷ $\mathcal{O}(1)$
10: $hayUnChocolate \leftarrow 0 \leq idChocolate < mapa.cantChocolates(m)$ ▷ $\mathcal{O}(1)$
11: **si** $hayUnChocolate$ **entonces** ▷ $\mathcal{O}(1)$
12: $\quad partida.chocolates[idChocolate] \leftarrow false$ ▷ $\mathcal{O}(1)$
13: $\quad partida.inmunidad \leftarrow 10$ ▷ $\mathcal{O}(1)$
14: **else**
15: $\quad partida.inmunidad \leftarrow 0$ ▷ $\mathcal{O}(1)$
16: $partida.perdio \leftarrow mapa.esCasilleroPeligroso(m, partida.jugador) \wedge partida.inmunidad = 0$ ▷ $O(1)$
17: $partida.gano \leftarrow false$ ▷ $O(1)$
18: $res \leftarrow partida$ ▷ $O(1)$

Complejdad: $\mathcal{O}(c)$
Justificación: La complejidad de este algoritmo es igual a recorrer el array $partida.chocolates$, que tiene longitud c, que es igual a la cantida de chocolates que contiene el mapa, para inicializar sus valores en $true$

---

iMover(**in/out** p: `partida`, **in** d: `direccion`)

1: **si** $d = ARRIBA$ **entonces** ▷ $\mathcal{O}(1)$
2: $desplazamiento \leftarrow (0, 1)$ ▷ $\mathcal{O}(1)$
3: **si** $d = ABAJO$ **entonces** ▷ $\mathcal{O}(1)$
4: $desplazamiento \leftarrow (0, -1)$ ▷ $\mathcal{O}(1)$
5: **si** $d = IZQUIERDA$ **entonces** ▷ $\mathcal{O}(1)$
6: $desplazamiento \leftarrow (-1, 0)$ ▷ $\mathcal{O}(1)$
7: **si** $d = DERECHA$ **entonces** ▷ $\mathcal{O}(1)$
8: $desplazamiento \leftarrow (1, 0)$ ▷ $\mathcal{O}(1)$
9: $nuevaPosicion \leftarrow p.jugador + desplazamiento$ ▷ $\mathcal{O}(1)$
10: **si** $\neg mapa.esPared(p.mapa, nuevaPosicion) \wedge mapa.enRango(p.mapa, nuevaPosicion)$ **entonces** ▷ $\mathcal{O}(1)$
11: $p.jugador \leftarrow nuevaPosicion$ ▷ $\mathcal{O}(1)$
12: $p.cantMov++$ ▷ $\mathcal{O}(1)$
13: $idChocolate \leftarrow mapa.idChocolate(p.mapa, p.jugador)$ ▷ $\mathcal{O}(1)$
14: $hayUnChocolate \leftarrow 0 \leq idChocolate < mapa.cantChocolates(p.mapa)$ ▷ $\mathcal{O}(1)$
15: **si** $hayUnChocolate \wedge_L p.chocolates[idChocolate]$ **entonces** ▷ $\mathcal{O}(1)$
16: $p.chocolates[idChocolate] \leftarrow false$ ▷ $\mathcal{O}(1)$
17: $p.inmunidad \leftarrow 10$ ▷ $\mathcal{O}(1)$
18: **else**
19: $p.inmunidad \leftarrow max(p.inmunidad - 1, 0)$ ▷ $\mathcal{O}(1)$
20: $partida.perdio \leftarrow mapa.esCasilleroPeligroso(m, partida.jugador) \wedge partida.inmunidad = 0$ ▷ $\mathcal{O}(1)$
21: $partida.gano \leftarrow mapa.llegada(m) = partida.jugador$ ▷ $\mathcal{O}(1)$

Complejidad: $\mathcal{O}(1)$
Justificación: No se recorre ninguna array, solo se accede a indices especificios. Los datos que provee el mapa, se encuentran guardados en arrays, que se complejidad de acceso es $\mathcal{O}(1)$, igualmente el acceso y modificación del array $partida.chocolates$ es $\mathcal{O}(1)$

---

iGanó?(**in** p: `partida`) $\rightarrow$ res : $bool$

1: $res \leftarrow p.gano$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

iPerdió?(**in** p: `partida`) $\rightarrow$ res : $bool$

1: $res \leftarrow p.perdio$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

iJugador(**in** p: `partida`) $\rightarrow$ res : $coordenada$

1: $res \leftarrow p.jugador$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

iCantMov(**in** p: `partida`) $\rightarrow$ res : nat

1: $res \leftarrow p.cantMov$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

Servicios usados

- Modulo Mapa

Modulo Fichín

Interfaz

se explica con: FICHÍN

géneros: `fichin`.

Operaciones básicas de fichin

NUEVOFICHIN(**in** m: `mapa`) $\rightarrow$ res : `fichin`
Pre $\equiv$ {true}
Post $\equiv$ {res = nuevoFichin(m)}
Complejidad: $O(1)$
Descripción: Genera un fichín
Aliasing: Recibe el mapa por referencia

NUEVAPARTIDA(**in/out** f: `fichin`, **in** j: `jugador`) $\rightarrow$ res : `bool`
Pre $\equiv$ {$f_0 = f$}
Post $\equiv$ {$res = \neg$alguienJugando?(f) $\wedge_L$ res $\Rightarrow_L$ f = nuevaPartida(f_0, j) }
Complejidad: $O(c)$
Descripción: Inicia una nueva partida

MOVER(**in/out** f: `fichin`, **in** d: `dirección`) $\rightarrow$ res : `bool`
Pre $\equiv$ {$f_0 = f$}
Post $\equiv$ {res = alguienJugando?(f) $\wedge_L$ res $\Rightarrow_L$ f = mover(f_0, d) }
Complejidad: $O(|J|)$ donde $|J|$ es el más largo de los nombres de los jugadores
Descripción: Mueve en la dirrección indicada

VERRANKING(**in** f: `fichin`) $\rightarrow$ res : `ranking`
Pre $\equiv$ {true}
Post $\equiv$ {res = ranking(f)}
Complejidad: $O(1)$
Descripción: Devuelve el ranking del fichín
Aliasing: Devuelve el ranking por referencia

OBJETIVO(**in** f: `fichin`) $\rightarrow$ res : `tupla<jugador, nat>`
Pre $\equiv$ {alguienJugando?(f) $\wedge$ def?(jugadorActual(f), ranking(f))}
Post $\equiv$ {res = objetivo(f) }
Complejidad: $O(J \cdot |J|)$ donde J es la cantidad de jugadores y $|J|$ es el más largo de los nombres de los jugadores
Descripción: Devuelve una tupla con el oponente y su puntaje

Implementación

Representación

fichín se representa con fch

donde fch es tupla($mapa$: mp,
$alguienJugando$: bool,
$jugadorActual$: string,
$partidaActual$: pt,
$ranking$: dicc(string, nat))

Invariante de representación

Rep : fch $\longrightarrow$ boolean

Rep(e) $\equiv$ True $\iff$
($e.alguienJugando \iff$ (longitud($e.jugadorActual$) $> 0 \wedge \neg e.partida.gano \wedge \neg e.partida.perdio$)) $\wedge$
($e.pt.gano \Rightarrow$
def?($e.jugadorActual$, $e.ranking$) $\wedge_L$
obtener($e.jugadorActual$, $e.ranking$) $\leq e.partida.cantMov$) $\wedge$
($e.mapa = e.pt.mapa$)

Función de abstracción

Abs : fch e $\longrightarrow$ partida {Rep(e)}

($\forall$ e : fch) Abs(e) $=_{\text{obs}}$ f: fichin | mapa(f) $= e.mapa \wedge$
alguienJugando(f) $= e.alguienJugando \wedge$
ranking(f) $= e.ranking \wedge$
$e.alguienJugando \Rightarrow_L$ partidaActual(f) $= e.partidaActual \wedge$
$e.alguienJugando \Rightarrow_L$ jugadorActual(f) $= e.jugadorActual$

Algoritmos

---

iNuevoFichin(**in** m : `mapa`) $\rightarrow$ res : fichin

1: $res \leftarrow \langle m, False, NULL, NULL, vacio \rangle$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

iNuevaPartida(**in/out** f : `fichin`, **in** a : `jugador`) $\rightarrow$ res : $bool$

$res \leftarrow False$
si $\neg f.alguienJugando$ **entonces**
$f.jugadorActual \leftarrow jugador$ ▷ $\Theta(1)$
$f.alguienJugando \leftarrow True$ ▷ $\Theta(1)$
$f.partidaActual \leftarrow partida.iNuevaPartida(f.mapa)$ ▷ $\mathcal{O}(c)$
$res \leftarrow True$

Complejidad: $\mathcal{O}(c)$
Justificación: La complejidad de este algoritmo es igual a la complejidad de $iNuevaPartida$, que es $\mathcal{O}(c)$, con $c = \#chocolates$.

---

iMover(in/out f : `fichin`, **in** d : `direccion`) $\rightarrow$ res : $bool$

1: **si** $f.alguienJugando$ **entonces**
2: partida.iMover(f.partidaActual, d) ▷ $\Theta(1)$
3: $res \leftarrow True$
4: **si** partida.iGanó?(f.partidaActual) **entonces** ▷ $\mathcal{O}(1)$
5: **si** def?(f.jugadorActual, f.ranking) **entonces** ▷ $\mathcal{O}(|J|)$
6: **si** obtener(f.jugadorActual, f.ranking) > partida.iCantMov(f.partidaActual) **entonces** ▷ $\mathcal{O}(|J|)$
7: definir(f.jugadorActual, partida.iCantMov(f.partidaActual), f.ranking) ▷ $\mathcal{O}(|J|)$
8: **else** definir(f.jugadorActual, partida.iCantMov(f.partidaActual), f.ranking) ▷ $\mathcal{O}(|J|)$
9: **else**
10: $res \leftarrow False$ ▷ $\Theta(1)$

Complejidad: $\mathcal{O}(|J|)$ si se realiza un movimiento que finaliza la partida
Justificación: El diccionario se implementa con un trie, que tiene $\mathcal{O}(|J|)$ como complejidad de def?, acceder a significado y definir.

---

iVerRanking(in f : `fichin`) $\rightarrow$ res : $dicc$

1: $res \leftarrow f.ranking$ ▷ $\Theta(1)$

Complejidad: $\Theta(1)$

---

iObjetivo(in f : `fichin`, **in** j : `jugador`) $\rightarrow$ res : $\langle jugador, nat \rangle$

1: $i \leftarrow 0$ ▷ $\Theta(1)$
2: $C \leftarrow obtenerClavesTrie(f.ranking)$ ▷ $\Theta(J)$
3: $S \leftarrow \emptyset$ ▷ $\Theta(1)$
4: **mientras** $i < long(C) - 1$ **hacer**
5: **si** $obtener(C[i], f.ranking) < obtener(jugador, f.ranking)$ **entonces** ▷ $\mathcal{O}(|J|)$
6: $Ag(C[i], S)$ ▷ $\Theta(1)$
7: $i \leftarrow i + 1$ ▷ $\Theta(1)$
8: **si** $\emptyset?(S)$ **entonces**
9: $res \leftarrow \langle jugador, obtener(f.ranking, jugador) \rangle$ ▷ $\Theta(1)$
10: **else**
11: $max \leftarrow obtener(f.ranking, S[0])$ ▷ $\mathcal{O}(|J|)$
12: $maxJugador \leftarrow S[0]$ ▷ $\Theta(1)$
13: $j \leftarrow 0$
14: **mientras** $j < long(S)$ **hacer**
15: **si** $obtener(f.ranking, S[j]) \geq max$ **entonces** ▷ $\Theta(|J|)$
16: $max \leftarrow obtener(f.ranking, S[j])$ ▷ $\Theta(|J|)$
17: $maxJugador \leftarrow S[j]$ ▷ $\Theta(1)$
18: $j \leftarrow j + 1$
19: $res \leftarrow \langle maxJugador, max \rangle$

Complejidad: $\sum_{k \in J} \mathcal{O}(|J|) + \mathcal{O}(|J|) + \sum_{k \in S} \mathcal{O}(|J|) = \mathcal{O}(|J| * J)$
Justificación: Los conjuntos C, S son implementados con una lista enlazada para que insertar elementos sea $\mathcal{O}(1)$.

Servicios usados

- Modulo Mapa
- Modulo Partida
- Modulo Diccionario Trie

Módulo Diccionario Trie(κ, σ)

El módulo Diccionario Trie provee un diccionario básico en el que se puede definir, borrar, y testear si una clave está definida en tiempo $\mathcal{O}(|J|)$, donde J es la cantidad de elementos del diccionario y $|J| = \text{máx}_{n \in J} |n|$.

Para describir la complejidad de las operaciones, vamos a llamar *copy*(k) al costo de copiar el elemento $k \in \kappa \cup \sigma$ y *equal*(k_1, k_2) al costo de evaluar si dos elementos $k_1, k_2 \in \kappa$ son iguales (i.e., *copy* y *equal* son funciones de $\kappa \cup \sigma$ y $\kappa \times \kappa$ en $\mathbb{N}$, respectivamente).[1]

Interfaz

parámetros formales

géneros κ, σ

función $\bullet = \bullet$(**in** $k_1 : \kappa$, **in** $k_2 : \kappa$) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} (k_1 = k_2)$}
Complejidad: $\Theta(equal(k_1, k_2))$
Descripción: función de igualdad de κ's

función COPIAR(**in** $k : \kappa$) $\rightarrow$ *res* : κ
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} k$}
Complejidad: $\Theta(copy(k))$
Descripción: función de copia de κ's

función COPIAR(**in** $s : \sigma$) $\rightarrow$ *res* : σ
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}} s$}
Complejidad: $\Theta(copy(s))$
Descripción: función de copia de σ's

se explica con: DICCIONARIO(κ, σ), ITERADOR BIDIRECCIONAL(TUPLA(κ, σ)).

géneros: `dicc`(κ, σ), `itDicc`(κ, σ).

Operaciones básicas de diccionario

VACÍO() $\rightarrow$ *res* : `dicc`(κ, σ)
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ vacio}
Complejidad: $\Theta(1)$
Descripción: genera un diccionario vacío.

DEFINIR(**in/out** d: `dicc`(κ, σ), **in** $k : \kappa$, **in** $s : \sigma$)
Pre $\equiv$ {$d =_{\text{obs}} d_0$}
Post $\equiv$ {$d =_{\text{obs}}$ definir(d, k, s)}
Complejidad: $\mathcal{O}(|J|)$, donde $|J|$ es el nombre más largo entre los jugadores
Descripción: define la clave k con el significado s en el diccionario.

DEFINIDO?(**in** d: `dicc`(κ, σ), **in** $k : \kappa$) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ def?(d, k)}
Complejidad: $\mathcal{O}(|J|)$, donde $|J|$ es el nombre más largo entre los jugadores
Descripción: devuelve `true` si y sólo k está definido en el diccionario.

SIGNIFICADO(**in** d: `dicc`(κ, σ), **in** $k : \kappa$) $\rightarrow$ *res* : σ
Pre $\equiv$ {def?(d, k)}
Post $\equiv$ {alias($res =_{\text{obs}}$ Significado(d, k))}
Complejidad: $\mathcal{O}(|J|)$, donde $|J|$ es el nombre más largo entre los jugadores
Descripción: devuelve el significado de una clave en el diccionario.

BORRAR(**in/out** d: `dicc`(κ, σ), **in** $k : \kappa$)
Pre $\equiv$ {$d = d_0 \wedge$ def?(d, k)}
Post $\equiv$ {$d =_{\text{obs}}$ borrar(d_0, k)}
Complejidad: $\mathcal{O}(|J|)$, donde $|J|$ es el nombre más largo entre los jugadores
Descripción: elimina la clave k y su significado de d.

[1] Nótese que este es un abuso de notación, ya que no estamos describiendo *copy* y *equal* en función del tamaño de k. A la hora de usarlo, habrá que realizar la traducción.

ObtenerClavesDelTrie(**in** d: `dicc(`κ, σ`)`) $\rightarrow$ res : `conj(`κ`)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ claves(d)}
Complejidad: $\Theta(J)$
Descripción: devuelve el conjunto de claves del diccionario.

#Claves(**in** d: `dicc(`κ, σ`)`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ #claves(d)}
Complejidad: $\Theta(1)$
Descripción: devuelve la cantidad de claves del diccionario.

Copiar(**in** d: `dicc(`κ, σ`)`) $\rightarrow$ res : `dicc(`κ, σ`)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ d}
Complejidad: $\Theta\left(\sum_{k \in K} (copy(k) + copy(\text{significado}(k, d)))\right)$, donde $K = \text{claves}(d)$
Descripción: genera una copia nueva del diccionario.

$\bullet = \bullet$(**in** d_1 : `dicc(`κ, σ`)`, **in** d_2 : `dicc(`κ, σ`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ $c_1 = c_2$}
Complejidad: $O\left(\sum_{\substack{k_1 \in K_1 \\ k_2 \in K_2}} equal(\langle k_1, s_1 \rangle, \langle k_2, s_2 \rangle)\right)$, donde $K_i = \text{claves}(d_i)$ y $s_i = \text{significado}(d_i, k_i)$, $i \in \{1, 2\}$.
Descripción: compara d_1 y d_2 por igualdad, cuando σ posee operación de igualdad.
Requiere: $\bullet = \bullet$(**in** $s_1 : \sigma$, **in** $s_2 : \sigma$) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ $(s_1 = s_2)$}
Complejidad: $\Theta(equal(s_1, s_2))$
Descripción: función de igualdad de σ's