Algoritmos y Estructuras de datos II - Trabajo Práctico 3

Grupo BlackMesa

Queda complejidades, aliasing, justificaciones sobre estructura y complejidad y cambiar las weas de nivel por las func de tuplas

1. Módulo Juego

Interfaz

se explica con: JUEGO
géneros: juego

Operaciones básicas de juego

NUEVOJUEGO(**in** *niveles*: `lista(nivel)`) $\longrightarrow$ *res*: `juego`
Pre $\equiv$ {tam(niveles) $\neq$ 0}
Post$\equiv$ {nivelesPendientes(res) $=_{obs}$ niveles $\wedge$ nivelActual(res) $=_{\text{obs}}$ nuevo$_S$(prim(niveles))}
Complejidad: $O(\sum_{i=0}^{long(l)}$ copy(nivel) + C + D^2 + P^2)
Descripcion: Creamos una nueva instancia en base a el conjunto de niveles dado.

MOVER(**in/out** *j*: `juego`, **in** *dir*: `nat`)
Pre $\equiv$ {j = j_0 $\wedge$ d $\in$ {0, 1, 2, 3} }
Post $\equiv$ {**if** puedeMover?(nivelActual(j_0),d) **then** j = mover(j_0,d) **else** j = j_0 **fi**}
Complejidad: si el movimiento no genera que el jugador gane entonces tiene que ser O(B + C + log P + log D). Si se gana el juego la complejidad es O(B + C + log(P) + log(D) + P^2 + D^2)
Descripccion: Mueve la posicion del jugador en la direccion especificada si se pudiese. En caso de que sea un movimiento ganador termina el nivel y genera una nueva partida con el siguiente nivel en la secuencia

TIRARBOMBA(**in/out** *j*: `juego`)
Pre $\equiv$ {j = j_0}
Post $\equiv$ { **if** #bombas(nivelActual(j) > 0 **then** j = tirarBomba(j_0) **else** j = j_0 **fi** }
Complejidad: O(1)
Descripccion: Pone una bomba en la posicion del jugador si las hay disponibles.

DESHACER(**in/out** *j*: `juego`)
Pre $\equiv$ {j = j_0}
Post $\equiv$ {j = deshacer(j_0)}
Complejidad: O(1)
Descripccion: Vuelve el nivel al estado inmediatamente anterior. En caso que el estado actual sea el inmediato al comienzo de un nivel se mantiene igual.

Representación

`Juego` se representa con `estr`

donde `estr` es `tupla`<*niveles*: `lista(nivel)`, *nivelActual*: `soko`>

Elegimos esta estructura para poder ofrecer una interfaz mas clara para el usuario con respecto a que acciones se pueden realizar en el nivel actual y poder manejar de mejor manera el cambio de los niveles, delegando el como se realizan estas acciones a un modulo de nivel inferior que denominamos Soko.

Invariante de Representación

Rep: $\hat{estr}$ e $\rightarrow$ boolean

Rep(e) $\equiv$ true $\iff$

$\{$mapa(volverAlInicio(e.nivelActual)) $=_{\text{obs}}$ $mapa_N$(prim(e.niveles)) $\wedge$

#bombas(volverAlInicio(e.nivelActual) $=_{\text{obs}}$ #bombas$_N$(prim(e.niveles)) $\wedge$

persona(volverAlInicio(e.nivelActual)) $=_{\text{obs}}$ persona$_N$(prim(e.niveles)) $\wedge$

$(\forall c$: coord)(c $\in$ cajas$_N$(prim(e.niveles)) $\iff$ hayCaja?(volverAlInicio(e.nivelActual), c)$\}$

volverAlInicio : soko $\longrightarrow$ soko

volverAlInicio(s) $\equiv$ **if** deshacer(s) $=_{\text{obs}}$ s **then** s **else** volverAlInicio(deshacer(s)) **fi**

Función de abstracción

Abs: $\hat{estr}$ e $\rightarrow$ juego

$(\forall$ e : $\hat{estr})$ Abs(e) $=_{\text{obs}}$ j : juego $\iff$

(nivelActual(j) $=_{\text{obs}}$ e.nivelActual) $\wedge$ $(\forall$ n:nivel) (n $\in$ e.niveles $\iff$ n $\in$ nivelesPendientes(j))

Algoritmos

NUEVOJUEGO (**in** *niveles*: `lista(nivel)`) $\longrightarrow$ *res*: `juego`

1: $res \leftarrow <$ $niveles$, NEWP(PRIMERO($niveles$)) $>$
2: **devolver** *res*

Analisis de complejidad: Esta funcion tendrá una complejidad igual al costo de copia de la lista de los niveles más el costo de la funcion NEWP

MOVER (**in/out** *estr*: `juego`, **in** *dir*: `nat`)

1: **si** MOV?(*estr.nivelActual, dir*) **entonces**
2: MOVJ(*estr.nivelActual, dir*)
3: **si** WIN?(*estr.nivelActual*) **entonces**
4: FIN(*estr.niveles*)
5: *estr.nivelActual* $\leftarrow$ NEWP(PRIMERO(*estr.niveles*))

Analisis de Complejidad: La guarda del primer condicional va a tener una complejidad O(B + C + log(P)), que es la complejidad de la funcion MOV?. El cuerpo del primer condicional tendra complejidad O(C + log(D)), que es la complejidad de la funcion MOVJ. En total el condicional tiene complejidad O(B + C + log(P)) + log(D), siendo esta la complejidad total de la funcion si no se realiza un movimiento ganador.
El segundo condicional tiene en su guarda complejidad O(1), y en su cuerpo complejidad O(C + P^2 + D^2). La complejidad entonces, si se gana el nivel nos queda O(B + C + log(P) + log(D) + P^2 + D^2)

TIRARBOMBA(**in/out** *estr*: `juego`)

1: **si** #TNT(*estr.nivelActual*) **entonces**
2: BOOM(*estr.nivelActual* $\neq$ 0)

Analisis de Complejidad: Tanto las funcion #TNT como la funcion BOOM tienen tiempo de peor caso constante, entonces la complejidad total de la funcion es O(1)

DESHACER (**in/out** *estr*: `juego`

1: UNDO(*estr.nivelActual*)

Analisis de complejidad: Como el acceso a el elemento de la tupla es contantes y el costo de la funcion UNDO es tambien O(1), la complejidad total de la funcion es O(1)

Servicios usados:

Este módulo utiliza el modulo **Lista Enlazada** del apunte de modulos basicos y el modulo **Soko** detallado más abajo.
El tipo `nivel` esta representado por
`tupla` $<$ *posij*: `coor`, *#bomba*: `nat`, *boxes*: `vector(coor)`, *walls*: `vector(coor)`, *depos*: `vector(coor)`, *docup*: `nat`$>$
El tipo `coor` esta representado por tupla$<$ x: `nat`, y: `nat`$>$

2. Módulo Soko

Interfaz

se explica con: SOKOBAN
géneros: soko

Operaciones básicas de soko

NEWP(**in** *nivel*: `nivel`) $\longrightarrow$ *res*: `soko`
Pre $\equiv$ {true}
Post$\equiv$ {persona(res) $=_{\text{obs}}$ persona$_N$($nivel$) $\wedge$
#bombas(res) $=_{\text{obs}}$ #bombas$_N$($nivel$) $\wedge$
depositos(mapa(res)) $=_{\text{obs}}$ depositos(mapa$_N$($nivel$)) $\wedge$
($\forall c$: coord)(c $\in$ cajas$_N$(prim(e.niveles)) $\iff$ hayCaja?(volverAlInicio(e.nivelActual), c) $\wedge$
($\forall c$: coord)(hayPared?(mapa(($res.ppar$, c)) $\iff$ hayPared?(mapa$_N$($nivel$), c) }
Complejidad: $O(C + D^2 + P^2)$
Descripcion: Creamos una nueva instancia para una partida de sokoban en base al nivel pasado por parametro.

MOVJ(**in/out** *s*: `soko`, **in** *dir*: `nat`)
Pre $\equiv$ {s = s_0 $=_{\text{obs}}$ puedeMover?(s, dir)}
Post $\equiv$ {s $=_{\text{obs}}$ mover(s_0, dir)}
Complejidad: O(C + log D)
Descripcion: en caso de que se posible mueve al jugador en la direccion especificada.

BOOM(**in/out** *s*: `soko`)
Pre $\equiv$ {$s = s_0 \wedge$ #bombas(s) $\neq 0$}
Post $\equiv$ {s $=_{\text{obs}}$ tirarBomba(s_0)}
Complejidad:O(1)
Descripcion: Pone una bomba en la posicion del jugador si las hay disponibles, destruyendo todas las paredes que se encuentren en la misma fila o columna.

UNDO(**in/out** *s*: `soko`)
Pre $\equiv$ {$s = s_0$}
Post $\equiv$ {deshacer(s_0)}
Complejidad:O(1)
Descripcion: Vuelve el nivel al estado inmediatamente anterior. En caso que el estado actual sea el inmediato al comienzo de un nivel se mantiene igual.

WIN?(**in** *s*: `soko`) $\longrightarrow$ *res*: `bool`
Pre $\equiv$ {True}
Post $\equiv$ {$res \iff$ gano?(s)}
Complejidad: O(1)
Descripcion: Devuelve verdadero si en la partida recibida todos los depositos tienen una caja sobre ellos.

#TNT(**in** *s*: `soko`) $\longrightarrow$ *res*: `nat`
Pre $\equiv$ {True}
Post $\equiv$ {res = #bombas(s) }
Complejidad:O(1)
Descripcion: Devuelve la cantidad de bombas que posee el jugador

MOV?(**in** s: `soko`, **in** dir: `nat`) $\longrightarrow$ res: `bool`
Pre $\equiv$ {True}
Post $\equiv$ $\{res \iff$ puedeMover?$(s,\ dir)\}$
Complejidad: O(B + C + log P)
Descripcion: Devuelve verdero si se pude mover al jugador en la direccion especificada.

SIGP(**in** c: `coor`, **in** dir: `nat`) $\longrightarrow$ res: `coor`
Pre $\equiv$ {True}
Post $\equiv$ {**if** $dir =_{\text{obs}} 0 \vee dir = 1$ **then**
$< \pi_1(c)$, **if** $dir = 0$ **then** $\pi_2(c) + 1$ **else** $\pi_2(c) - 1$ **fi** $>$
else
<**if** $dir = 2$ **then** $\pi_1(c) + 1$ **else** $\pi_1(c) - 1$ **fi** $>$, $\pi_2(c)$
fi}
Complejidad: O(1)
Descripcion: dada una posicion devuelve la siguiente posicion en la direccion especificada.

PAR?(**in** s: `soko`, **in** c: `coor`) $\longrightarrow$ res: `bool`
Pre $\equiv$ {True}
Post $\equiv$ $\{res \iff$ hayPared?(mapa(s), c)}
Complejidad:O(B + log(P))
Descripcion: Devuelve verdero si hay una pared en la posicion

Representación

`Soko` se representa con `estr`
donde `estr` es `tupla`< *posj*: `coor`, *TNTs*: `nat`, *pbox*: `lista(coor)`, *pTNT*: `lista(coor)`,
ppar: `vector(coor)`, *pdep*: `vector(coor)`, *docu*: `nat`,
acts: `pila(tupla`< *movi*: `nat`, *box?*: `bool`, *dep?*: `int`>)>

Elegimos esta estructura pensando en como resolver las complejidades pedidas. Para las paredes y los depositos utilizamos vectores para poder tener coordenadas ordenadas siguiendo una relacion de orden arbitraria, que nos permitiera buscar elementos en tiempo logaritmico utilzando busqueda binaria.

Para las cajas y las bombas elegimos usar listas en cambio debido a que la mayor parte de la funciona provistas en este modulo para manipular las misma no ofrecian complejidad constantes que nos ayudaron a la hora de plantear algoritmos para deshacer movimientos y poner bombas.

Elegimos utilizar una pila para representar las acciones utilizadas, donde los elementos de la pila son tuplas que contiene un nat que representa la dirrecion en la que se realizo el movimiento anterior o el numero 4 que representa el uso de una bombar. Además la tupla contiene dos booleanos que nos ayudan a ver si en el movimiento anterior se movieron cajas y si se ocupo algun deposito facilitando las operaciones en la funcion undo para mantener su costo constante.

Por último elegimos utilizar un natural para representar la cantida de depositos ocupados para tener una manera de poder verificar si se gano el juego en tiempo constante.

Invariante de Representación

1. La posicion del jugador debe ser distinta a la de todas las cajas y a la de todas las paredes, a menos que se haya puesto una bomba en alguna coordenada que comparta componente con la coordenada de la pared en cuestion.
2. Las coordendas de las cajas no pueden coincidir con las de las paredes salvo, nuevamente, que esta pared este en alguna posicion que comparta componente con una posicion en donde se haya arrojado una bomba.
3. Las posiciones de los depositos no pueden coincidir con las posiciones de las paredes.
4. Las coordenadas de los depositos y las paredes deben estar ordenadas, siguiendo la siguiente relacion de orden: se comparan las primeras componentes y en caso de ser iguales se define comparando las segundas coordenadas.
5. La cantidad de depositos ocupados no puede ser mayor que la cantidad de depositos que hay en el juego.
6. La pila solo puede contener numeros naturales que vayan desde el 0 hasta el 4 inclusive como primer componente de las tuplas y su ultima componente solo puede contener numeros del -1 al 1.
7. El largo del vector de bombas tiene que coincidir con la cantidad de apariciones del numero 4 en la pila de acciones.

Función de abstracción
Abs: $\hat{estr}$ e $\rightarrow$ soko
($\forall$ e : $\hat{estr}$) Abs(e) $=_{\text{obs}}$ s : soko $\iff$
$estr.posj =_{\text{obs}}$ persona(s) $\wedge$ $estr.TNTs =_{\text{obs}}$ #bombas(s) $\wedge$
($\forall c$: coord)(esta?($estr.pTNT$, c) $\Rightarrow$
($\nexists c_1$: coord)(($\pi_1(c) =_{\text{obs}} \pi_1(c_1) \vee \pi_2(c) =_{\text{obs}} \pi_2(c_1)$) $\wedge$ hayPared?(mapa(s), c_1)) $\wedge$
($\forall c$: coord)(esta?($estr.ppar$, c) $\wedge$ ($\nexists c_1$: coord)(esta?($estr.TNTs$, c_1) $\wedge(\pi_1(c) =_{\text{obs}} \pi_1(c_1) \vee \pi_2(c) =_{\text{obs}} \pi_2(c_1))$
$\iff$ hayPared?(mapa(s), c)) $\wedge$
($\forall c$: coord)(esta?($estr.pbox$) $\iff$ hayCaja?(s, c)) $\wedge$
($\forall c$: coord)(esta?($estr.pdep$,c) $\iff$ c $\in$ depositos(mapa(s)))

Algoritmos

NEWP (**in** *nivel*: `nivel`) $\longrightarrow$ *res*: `soko`

$res \leftarrow <$ *nivel.posij*, *nivel.#bomb*, *nivel.boxes*, VACIA(), ORDENAR(*nivel.walls*, ORDENAR(*nivel.depos*, *nivel.docup*, VACIA() $>$
devolver *res*

Analisis de complejidad: la complejidad de este algoritmos es el la suma de las complejidades del costo de copia de los elementos que se pasan desde el nivel (posicion inicial(O(1)), bombas inciales(O(1)), cajas(O(C)), paredes (O(P)), depositos(O(D)), depositos ocupados (O(1))) más el costo de complejidad de la funcion ORDENAR(O(P^2) y O(D^2)). Finalmente la complejidad queda O(C + D^2 + P^2)

```
MOVJ(in/out s: soko, in dir: nat)
s.posj ← SIGP(s.posj, dir)
aux1 ← False
aux2 ← 0
it ← CREARIT(s.pbox)
para i = 0 ... LONGITUD(s.pbox) - 1 hacer
    si SIGUIENTE(it) = s.posj entonces
        ELIMINAR(SIGUIENTE(it))
        AGREGARATRAS(S.PBOX, SIGP(s.posj, dir))
        aux1 ← True
    AVANZAR(it)
si aux1 y PERTENECELOG(SIGP(s.posj, dir), s.pdep) entonces
    s.docu ← s.docu + 1
    aux2 ← −1
si aux1 y PERTENECELOG(s.posj), s.pdep) entonces
    s.docu ← s.docu − 1
    aux2 ← aux2 + 1
APILAR(s.acts, <dir, aux1, aux2>)
```

Analisis de Complejidad: Las cuatro primeras lineas y la ultima son constantes debido a que son asignaciones y usan otras funciones de complejidad constante.
El cuerpo del ciclo va a tener complejidad O(1), debido a que utiliza operaciones y funciones que tienen costo constante de los modulos iterador de lista y lista enlazada. Como este ciclo se repite una cantidad de veces igual a la longitud de la lista de las cajas la complejidad del mismo queda O(C)
Por ultimo, en los dos condicionales se aplica en las guardas la funcion PERTENCELOG sobre el arreglo ordenado de los depositos, que tiene complejidad de peor caso O(log(D)). El resto de las operaciones de los mismos son operacionse de costo constante.
Finalmente la complejidad queda O(C+Log(D))

```
BOOM (in/out s: soko)
AGREGARATRAS(s.pTNT, s.posj)
s.TNTs ← s.TNTs − 1
APILAR(s.acts, <4, False, 0>)
```

Como las funciones AGREGARATRAS y APILAR tienen costo O(copy(a)) que es el costo de copia de una tupla que es O(1) y la otra linea de codigo es una asignacion entonces la funcion tiene complejidad O(1)

```
WIN? (in s: soko) ⟶ res: bool
devolver s.docu = LONGITUD(s.pdep)
```

Como la funcion LONGITUD del modulo vector es O(1) la función tiene complejidad O(1)

```
#TNT (in s: soko) ⟶ res: nat
devolver s.TNTs
```

UNDO (**in/out** *s*: soko)

```
s.docu ← s.docu + TOPE(s.acts).dep?
si TOPE(s.acts).movi = 4 entonces
    COMIENZO(s.pTNT)
    s.TNTs ← s.TNTs + 1
else
    si TOPE(s.acts).box? entonces
        COMIENZO(s.pbox)
        AGREGARATRAS(s.pbox, s.posj)
    si TOPE(s.acts).movi <2 entonces
        si TOPE(s.acts).movi = 0 entonces
            s.posj ← SIGP(s.posj, 1)
        else
            s.posj ← SIGP(s.posj, 0)
    else
        si TOPE(s.acts).movi = 2 entonces
            s.posj ← SIGP(s.posj, 3)
        else
            s.posj ← SIGP(s.posj, 2)
```

Analisis de complejidad: Las fuciones TOPE, COMIENZO, AGREGAATRAS, SIGP son todas funciones cuya complejidad de peor caso es O(1). Como no hay ciclos y el resto de las operaciones son asignaciones o sumas o restas, que se resuelven en tiempo constante, la complejidad de la funcion queda O(1)

SIGP (**in** *c*: coor, **in** *dir*: nat) ⟶ *res*: coor

```
si dir <2 entonces
    si dir = 0 entonces
        dir.y ← dir.y + 1
    else
        dir.y ← dir.y − 1
else
    si dir = 2 entonces
        dir.x ← dir.x + 1
    else
        dir.x ← dir.x − 1
```

Analisis de complejidad: Como todas las operaciones de esta función son asginaciones o comparaciones elementales o sumas o restas sin ningun ciclo la complejidad de la funcion es O(1)

MOV? (**in** *s*: soko, **in** *dir*: d) ⟶ *res*: bool

```
devolver (PERTENECELINEAL(SIGP(s.posj, dir), s.pbox) ∧ ( PERTENECELINEAL(SIGP(SIGP(s.posj, dir), dir),
    s.pbox)) ∨ PAR?( s, SIGP(SIGP(s.posj, dir), dir))) ∨ PAR?(s, SIGP(s.posj, dir))
```

Analisis de complejidad: Como SIGP tiene complejidad constante la funcion PERTENECELINEAL tendrá complejidad de peor caso O(long(s.pbox)), es decir O(C). Luego PAR? tiene complejidad O(B + log(P)), entonces la complejidad final queda O(B + C + log(P))

PAR? (**in** s: `soko`, **in** c: `coor`) $\longrightarrow$ res: `bool`

si PERTENCELOG(c, $s.ppar$) **entonces**
    $aux \leftarrow s.pTNT$
    mientras LONGITUD($aux > 0$) **hacer**
        si $c.x$ = PRIMERO(aux).x $\vee$ $c.y$ = PRIMERO(aux).y **entonces**
            devolver False
        else
            FIN(aux)
    devolver True
else
    devolver False

Analisis del peor caso: la guarda del primer condicional tendría complejidad O(log(P)) siendo P la longitud de *s.ppar*. La asignacion de *aux* tendria como plejidad O(copy(s.pTNT)) que sería O(B) siendo la la longitud de s.pTNT .
El cuerpo del ciclo tendría complejidad del peor caso O(1) debido a que los accesos a elementos de tuplas y las funciones PRIMERO y FIN son O(1). Como el peor sería no encontrar ninguna coordenada en p.TNT que coincida en al menos una compenente con la coordenada pasada por parametro el ciclo se repetiria un total de veces igual a la longitud s.pTNT siendo su complejidad O(B)
Finalmente la complejidad de la funcion queda O(log(P)+ B)

Servicios usados:

Este módulo utiliza los modulos **Lista Enlazada, Vector, Pila e Iterador de lista** del apunte de modulos basicos.
El tipo `nivel` esta representado por
`tupla` < *posij*: `coor`, *#bomba*: `nat`, *boxes*: `vector(coor)`, *walls*: `vector(coor)`, *depos*: `vector(coor)`, *docup*: `nat`>
El tipo `coor` esta representado por tupla< *x*: `nat`, *y*: `nat`>

Además extendemos el módulo de **Vector** para hacer uso de las siguientes funciones de vectores:

- ORDENAR
- PERTENECELOG
- PERTENECELINEAL

Las funciones de pertenencia lineal y logaritmica utilizan la busqueda lineal(O(n)) y binaria(O(log(n))) respectivamente. Esto se hizo para poder obtener complejidades logaritmicas a la hora de buscar elementos en los vectores de paredes y despotiso con tal de cumplir con las especificaciones pedidas.

A continuacion detallaremos la funcion ORDENAR que utilizamos a fin de poder obtener, en base a un criterio de orden entre tuplas de int propio, arreglos de coordenadas ordendos en donde pudieramos utilizar la funcion de pertenencia logaritmica.

ORDENAR(**in** v: `vector(tupla<nat, nat>)`) $\longrightarrow$ res: `vector(tupla<nat, nat>)`
Pre $\equiv$ {True}
Post $\equiv$ { $(\forall c\text{: coor})(\text{esta?(res, c)} \iff \text{esta?}(v\text{, c})) \wedge$
$(\forall i\text{: nat})(0 \leq \text{i} \leq \text{long(res) - 2} \Rightarrow \pi_1(\text{res[i]}) < \pi_1(\text{res[i+1]}) \vee$
$(\pi_1(\text{res[i]}) = \pi_1(\text{res[i+1]}) \wedge \pi_2(\text{res[i]}) < \pi_2(\text{res[i+1]})))\}$
Complejidad: O(n^2)
Descripcion: Ordena el vector que recibe comparando las tuplas primero por su primer componente y en caso de ser iguales por su segunda componente

ORDENAR (**in** v: `vector(tupla<nat, nat>)`) $\longrightarrow$ *res*: `vector(tupla<nat, nat>)`

$res \leftarrow$ VACIA()
mientras LONGITUD(v) $>$ 0 **hacer**
    $min \leftarrow$ BUSCARMIN(v)
    AGREGARATRAS(RES, V[MIN])
    ELIMINAR(v, min)
devolver res

BUSCARMIN sería otra funcion del modulo vector que funciona solo para vectores de tuplas de coordenadas y nos devuelve el índice de el minimo elemento y tiene complejidad O(long(v)).
La funcion AGREGARATRAS tiene complejidad O(f(long(v)) + copy(a)) siendo que el costo de copia de la tupla y la f(long(v)) es O(1).
La complejidad de ELIMINAR es (long(v)-i) es decir O(n) en el peor caso. Así el cuerpo de ciclo queda con complejidad O(n) y debido a que este se repite una cantidad de veces igual a la longitud del vector el ciclo tiene una complejidad total de O(n^2).
El resto de las lineas tiene complejidad O(1).