Práctica 8
Interpretación de Enunciados Complejos

Introducción a la Programación
1er Semestre de 2017

Nota: la presente práctica se compone por enunciados de parciales de años anteriores y su objetivo es entrenar al alumno en la resolución de enunciados complejos.

1. Juego de la Vida

El *Juego de la Vida* consiste en simular la evolución de ciertas *células* que habitan el universo y que interactúan entre sí a lo largo del tiempo. Una célula puede estar *viva* o *muerta*. Como producto de esta interacción algunas células mueren y otras nuevas nacen. La interacción se produce entre células vivas que son *vecinas*. Una célula se dice vecina de otra si linda con la misma. La interacción entre células vivas sigue las siguientes *pautas de evolución*:

1. Toda célula viva con menos de dos vecinas vivas muere (escasez de población).
2. Toda célula viva con más de tres vecinas vivas muere (sobrepoblación).
3. Toda célula viva con exactamente dos o tres vecinas vivas pasa a la siguiente generación.
4. Toda célula muerta con exactamente tres vecinas vivas se convierte en una célula viva (reproducción).

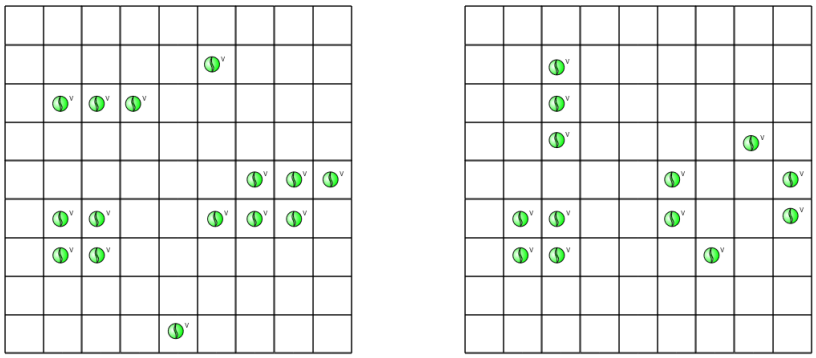
Estas pautas de juego se aplican *simultáneamente* sobre todas las células del universo. Cada una de estas aplicaciones simultáneas de las pautas de juego se conoce como un *tick*. Cada *tick* produce un nuevo universo.

El *Juego de la Vida* se puede modelar en Gobstones de la siguiente manera. Cada célula del juego puede representarse con una *celda* del tablero. La misma tiene una bolita verde si la célula que representa está viva y ninguna bolita verde si está muerta. Una célula es vecina de otra si sus correspondientes celdas son lindantes al N, NE, E, SE, S, SO, O u NO. La figura 1(a) muestra un universo posible y 1(b) exhibe cómo evoluciona en un tick.

Ejercicio 1

Escribir las siguientes funciones:

1. `celulaViva`, que determine si una célula está viva o no.
2. `nroVecinas`, que determine el número de células vivas que son vecinas de la célula actual.



(a) (b)

Figura 1: Ejemplo de universo y cómo evoluciona luego de un tick

Ejercicio 2

Escribir `ProcesarCelula`, un procedimiento que aplique las pautas de juego a la célular actual.

Nota: Tener en cuenta que no se puede eliminar o reanimar una célula hasta tanto no se hayan aplicado las pautas de juego a **todas** las células del universo, pues sino las células de la nueva generación se confundirían con aquellas de la etapa actual, haciendo que se produzcan resultados erróneos al procesar las pautas. En consecuencia, en lugar de eliminar o reanimar una célula, la misma debe ser **marcada** como que va a ser eliminada o reanimada (una vez que se complete el procesamiento de las demás células). Para ello, puede asumir que cuenta con procedimientos `MarcarCelulaParaSerEliminada()` y `MarcarCelulaParaSerReanimada()` que marcan una célula para ser eliminada o reanimada más tarde, respectivamente.

Ejercicio 3

Escribir `ActualizarUniverso`, un procedimiento que lee todas las marcas de eliminación/-reanimación de células y las procesa (es decir, efectivamente elimina/reanima esas células). El resultado será un nuevo universo, en condiciones de poder evolucionar en un tick subsiguiente.
Nota: Estructurar su solución como un recorrido. Puede asumir la existencia de:

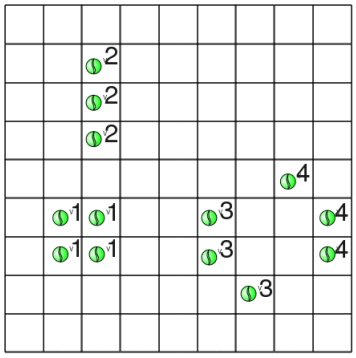
- Funciones `estaMarcadaParaSerEliminada()` y `estaMarcadaParaSerReanimada()`, que retorna un booleano indicando si esa célula está marcada para ser eliminada o reanimada, respectivamente.
- Procedimiento `EliminarCelula()` y `ReanimarCelula()`, que eliminan una célula si había allí una marca para eliminar y reaniman una célula si había allí una marca para reanimar. Además, estas operaciones quitan las marcas.

Ejercicio 4

Escribir `Simular`, un procedimiento que dado un número entero positivo, simula esa cantidad de ticks sobre el universo.

Habiendo hecho una simulación, es de interés contabilizar la cantidad de *islotes* en un universo, una vez finalizada la simulación. Un islote es un conjunto de células vivas que están *conectadas*. Dos células vivas se dicen conectadas si hay una secuencia de células vivas lindantes entre ellas. Por ejemplo, en la figura 1(a) hay 5 islotes mientras que en 1(b) hay 4.

Para contabilizar la cantidad de islotes es conveniente marcarlos[1], marcando todas las células vivas del mismo con, digamos, bolitas rojas. Es importante que dos islotes diferentes tengan marcas diferentes. A modo de ejemplo, el resultado de marcar todos los islotes del universo de la figura 1(b) podría ser:



Ejercicio 5

Escribir `hayCelulasSinMarcar`, una función que determine si hay células activas sin marcar.

Ejercicio 6

Escribir `computarNumeroDeIslotes`, una función que compute la cantidad de islotes que hay en el universo actual. Asumir la existencia de un procedimiento `MarcarIslote`, que dado un número positivo n y asumiendo que hay células sin marcar en el universo, marca el islote n-ésimo (colocando n bolitas rojas en cada célula del islote).

2. Buscaminas

Buscaminas es un juego que se juega sobre un tablero de *locaciones*. Cada locación puede estar tapada o destapada; también puede contener minas; al comienzo todas las locaciones están tapadas. El objetivo es adivinar dónde se encuentran ubicadas las minas. El problema es que las locaciones tapadas no dejan ver si hay una mina en la misma. Si se destapa una locación con una mina, la mina explota y se termina el juego. Por ello, hay que destapar locaciones que *no* tengan minas.

Una *partida* consiste en una serie de *jugadas*. Cada una indica una locación a destapar. Para evitar tener que seleccionar locaciones a ciegas, el juego brinda un mecanismo de asistencia: las locaciones pueden tener un número denominado *pista*. El mismo indica la cantidad de minas que hay en locaciones vecinas (N, NE, E, SE, S, SO, O, NO).

Cada celda del tablero puede verse como una locación. Al comienzo están todas tapadas. Se utilizan los siguientes códigos de colores para las locaciones: `Negro` (1=tapada; 0=destapada), `Rojo` (1=hay mina, 0=no hay mina) y `Azul` (pistas – 1 a 8 bolitas).

[1] Esta marca no guarda relación la marca para matar o reanimar células ya vista.

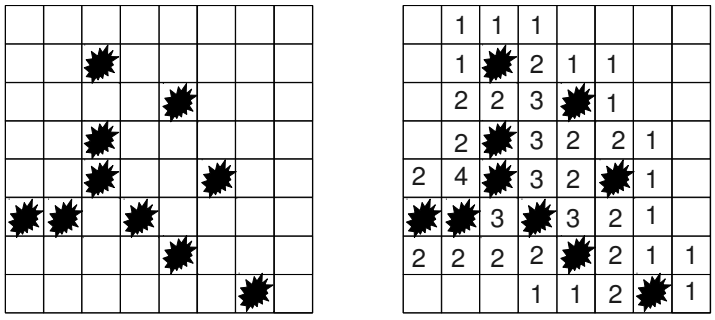


Figura 2: Ejemplo de tablero sin pistas y con pistas

Ejercicio 7

Escribir:

- `contarMinasVecinas`, una función que retorna un número indicando el total de las minas que hay en locaciones vecinas.
- `PonerPista`, un procedimiento que pone una pista en la locación actual. Asumir que en la locación actual no hay pista ni mina.

Ejercicio 8

Escribir `PonerPistas`, un procedimiento que pone las pistas en todas las locaciones del tablero (salvo aquellas que tienen minas). Puede asumir que hay al menos una locación con mina. Por ejemplo, dado el tablero izquierdo de la Fig. 2, debe arrojar como resultado el tablero derecho de esa figura.

Las pistas son una ayuda importante a la hora de saber si una locación tiene o no una mina. Como consecuencia de la información brindada por las pistas, para cada locación hay tres posibles juicios que pueden hacerse:

1. Seguro tiene una mina (STM).
2. Seguro no tiene mina (SNTM).
3. No se puede saber nada.

El objetivo de lo que sigue es escribir un procedimiento `InferirInfo` que analice las pistas y marque aquellas que STM y aquellas que SNTM. Esto permite al jugador tomar una decisión más informada sobre qué locación a destapar. Para poder definir `InferirInfo`, primero vamos a introducir algunos procedimientos y funciones auxiliares.

Nota: En lo que sigue se asume que las pistas ya fueron colocadas sobre el tablero.

Ejercicio 9

Escribir un procedimiento `ProcesarLocaciónConPista` que revise las locaciones vecinas y marque aquellas que seguro tienen una mina (STM) con una bolita verde y aquellas que seguro no tienen una mina (SNTM) con dos bolitas verdes. Asumir que el cabezal se encuentra sobre

una locación destapada y con pista. **Ayuda:** Supongamos que la pista de la locación actual es n:

- Si hay exactamente n locaciones vecinas tapadas (sin importar si son STM o SNTM), todas las locaciones vecinas deben ser marcadas como STM (si es que ya no están marcadas como STM).
- Si hay n locaciones vecinas tapadas y marcadas como STM, entonces *todas* las demás locaciones vecinas (si las hubiere) se marcan como SNTM.

Puede asumir la existencia de una función `nroDeVecinasTapadas` que retorna un número indicando la cantidad de locaciones vecinas que están tapadas; y `nroDeVecinasMarcadasConSTM` que retorna un número indicando la cantidad de locaciones vecinas que tienen la marca de STM.

Ejercicio 10

Escribir un procedimiento `ProcesarTodasLasLocacionesDelTablero` que procese todas las locaciones del tablero.

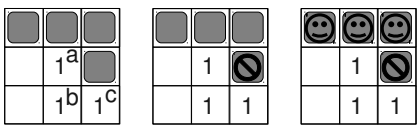
Ejercicio 11

Escribir las siguientes funciones:

- `nroDeMarcasSTMoSNTMEnTablero` que retorna un número indicando el total de marcas (tanto STM como SNTM) que hay en el tablero.
- `hayNuevasMarcas` que determine si el procedimiento `ProcesarTodasLasLocacionesDelTablero` coloca alguna marca STM o SNTM nueva en el tablero procesado respecto al tablero a procesar.

Ejercicio 12

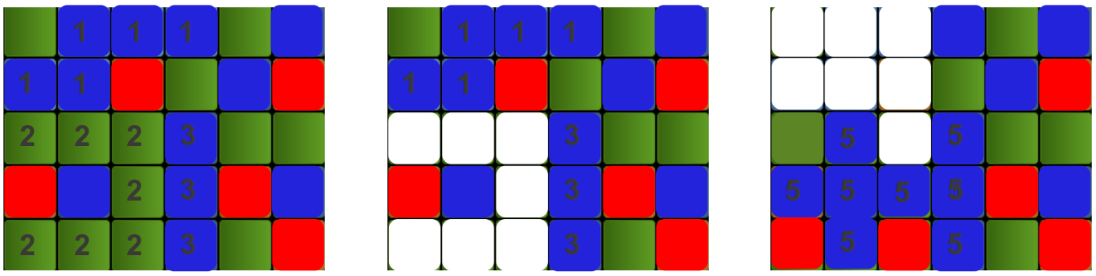
Escribir un procedimiento `InferirInfo` que procese todas las locaciones con pistas poniendo las marcas de STM y SNTM hasta que no haya posibilidad de poner más marcas. Es **fundamental** observar que una vez que se procesen todas las locaciones del tablero, es posible que se presenten nuevas oportunidades para colocar marcas. Por ejemplo, en el tablero de abajo a la izquierda, asumiendo que las locaciones con pistas se recorren en el orden indicado con a,b,c, el tablero resultante sería el del medio. Notar que se ha marcado una locación como que STM. Ahora, si volvemos a procesar el tablero del medio, nos queda el del extremo derecha. Notar que las tres locaciones de la primera fila ahora están marcadas como que no tienen minas (SNTM – la carita).



3. SameGame

SameGame es un juego de un solo jugador que se juega sobre un tablero rectangular de casilleros. Inicialmente, cada casillero tiene un color (azul, negro, rojo o verde). Una *pieza* consiste de un grupo de *al menos* tres casilleros que están **unidos** y son todos del **mismo color**. Más precisamente, dos casilleros forman parte de la misma pieza cuando se puede ir desde uno hacia el otro recorriendo *exclusivamente* celdas del mismo color (con movimientos hacia el Norte, Este, Sur u Oeste). Notar que los casilleros que **no forman parte** de la pieza y son vecinos de algún casillero que **sí forma parte** de la pieza deben tener un color distinto a los casilleros de la pieza.

La Figura 3 exhibe un tablero ejemplo. El mismo tiene un total de tres piezas, indicadas con números (nota: los números no forman parte del tablero). El jugador selecciona una pieza cualquiera del tablero. Una vez hecho esto, los casilleros que componen la pieza explotan permitiendo *caer* a los casilleros que estaban encima de la pieza. Notar que a medida que las piezas explotan, algunos casilleros dejan de tener color. El jugador obtiene un *puntaje* por cada pieza que explota, que depende de la cantidad de celdas que componen la pieza explotada. El objetivo del juego es obtener el mayor puntaje posible. El juego termina cuando el tablero queda totalmente vacío o ya no hay piezas para seleccionar.



(a) (b) (c)

Figura 3: Tablero ejemplo con todas las piezas marcadas

Se utilizará el tablero, las celdas y las bolitas de GOBSTONES para representar el tablero y los casilleros de *SameGame*. Por ejemplo, un casillero de color azul se representa con una celda con una bolita azul. Además, algunos casilleros pueden estar *marcados*. Los casilleros marcados se representan con **dos** bolitas del color del casillero. Por ejemplo, un casillero marcado de color rojo se representa con dos bolitas rojas.

Ejercicio 13

Escribir una función `esPieza` que retorna un booleano indicando si el casillero actual pertenece a una pieza. Para ello, alcanza con determinar si se verifica **cualquiera** de las siguientes condiciones:

1. El casillero actual tiene dos vecinos del mismo color que él.
2. El casillero actual tiene un vecino que cumple con el ítem anterior y tiene el mismo que el casillero actual.

Nota: puede asumir la existencia de las funciones `nroVecinos` que, dado un color c, denota la cantidad de celdas vecinas que tienen bolitas de color c, y `direccionConColor` que, dado un color c, denota la dirección de algún vecino que tienen bolitas de color c. La función `direccionConColor` es parcial; su precondición es que `nroVecinos(`c`)` > 0.

Ejercicio 14

Escribir un procedimiento `BuscarCasilleroMarcado` que posiciona el cabezal de GOBSTONES sobre el casillero que esté marcado. **Nota:** puede suponer que hay al menos un casillero marcado en el tablero.

Dado que una pieza se compone de casilleros, también puede marcarse una *pieza*. Para ello basta marcar cualquiera de los casilleros que la componen.

Ejercicio 15

Escribir un procedimiento `ExplotarPiezaMarcada` que haga explotar la pieza que se encuentra marcada. Hacer explotar la pieza implica hacer explotar todos los casilleros que la componen. Por ejemplo, si la pieza marcada en la figura 3(a) es la 2, entonces el tablero resultando sería el de la Figura 3(b). En consecuencia el procedimiento debe:

1. Buscar el casillero marcado.
2. Marcar los vecinos que tengan el mismo color.
3. Explotar el casillero actual.
4. Repetir a)-c) hasta que no queden casilleros marcados.

Nota: puede asumir la existencia de la función `hayCasilleroMarcado` que indica si hay al menos un casillero marcado en el tablero, y la existencia de los procedimientos `MarcarCasillerosVecinos` que marca los casilleros vecinos al actual que tienen el mismo color y `ExplotarCasillero` que explota un casillero (i.e. deja la celda que lo modela vacía).

Ejercicio 16

Escribir una función `puntajeDePiezaMarcada` que retorna el puntaje que se obtendría si se hiciera explotar la pieza actualmente marcada. El puntaje a retornar es $n \times (n-1)$ donde n es la cantidad de casilleros de la pieza. **Nota:** Puede hacer uso del ejercicio 9 de la práctica 2 (el mismo solicita realizar un procedimiento que cuente la cantidad de bolitas que haya en el tablero de un color dado).

Ejercicio 17

Escribir un procedimiento `AplicarFuerzaDeGravedadACasillero` que mueve el casillero actual hacia el sur simulando la caída del casillero como efecto de la fuerza de gravedad. Para ello, la bolita del casillero actual debe moverse al sur, tantas veces como sea posible, sin que dos bolitas compartan el mismo casillero. **Nota:** puede asumir que el casillero actual no está vacío.

Ejercicio 18

Escribir un procedimiento `AplicarGravedadAlTablero` que aplica la fuerza de gravedad a todos los casilleros, de forma tal que ningún casillero quede "flotando". Por ejemplo, si se

explota la pieza 2 de la Figura 3(a), entonces el tablero resultando sería el de la Figura 3(c). El nuevo tablero tiene sola una pieza (que además es nueva).

Suponga que dispone de un procedimiento `InteligenciaArtificial`, que no tiene parámetros, cuyo objetivo es proveer una estrategia de juego. Para ello, `InteligenciaArtificial` marca la pieza del tablero que se debe explotar de acuerdo a la estrategia. Como precondición, ninguna pieza puede estar marcada y el juego no debe haber *finalizado*, i.e. aún deben quedar piezas en el tablero.

Ejercicio 19

Escribir una función `simularJuego` que simula el juego con la estrategia de la inteligencia artificial y retorna el puntaje conseguido. **Nota:** puede suponer que ninguna pieza se encuentra marcada y que existe una función `juegoFinalizado` que devuelve un booleano indicando si quedan piezas.

4. Nurikabe

NURIKABE es un juego en el que un jugador debe dibujar un río en un mapa de forma tal de generar un conjunto de islas que satisfagan ciertas particularidades. El *mapa* es una grilla rectangular formada por celdas. Inicialmente algunas de estas celdas contienen un número y el resto se encuentran vacías (ver Figura 4(a)). Las celdas con número se llaman *indicadoras*. A medida que el juego transcurre, el jugador va *pintando* algunas de las celdas vacías con un lápiz negro; estas celdas negras van formando los segmentos del río, y dan origen a la formación de islas. Una *isla* es simplemente una región sin celdas negras[2] que está bordeada por el río o por los bordes del mapa (ver Figura 4(b)). El objetivo del juego es dibujar el río de forma tal que al finalizar se satisfagan las cuatro condiciones siguientes (ver Figuras 4(c) y 4(d)).

- Cada isla tiene una única celda indicadora (el resto de sus celdas están vacías).
- Cada isla tiene tantas celdas como el número que indica su única celda indicadora.
- El río es continuo, es decir, se debe poder recorrer todas las celdas negras con movimientos horizontales y verticales sin pasar por celdas blancas.
- No hay ninguna región de 2×2 celdas que contenga sólo celdas negras.

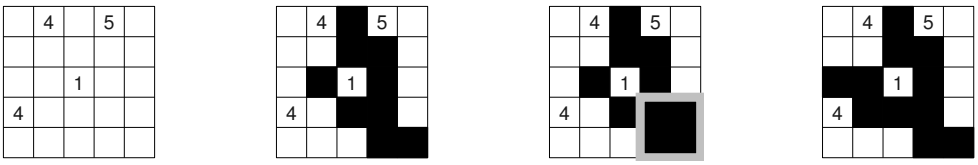
Para representar el mapa vamos a utilizar el tablero. Cada celda pintada se representa poniendo una bolita negra, las celdas indicadoras tienen tantas bolitas azules como el número correspondiente, y las celdas vacías no tienen bolitas (ver Figura 5).

Para simplificar la tarea de programar el juego, nos conviene tener un procedimiento que nos permita marcar cada isla del mapa, usando bolitas **rojas**. Decimos que una celda está *marcada* si contiene una bolita roja y *desmarcada* en caso contrario.

Ejercicio 20

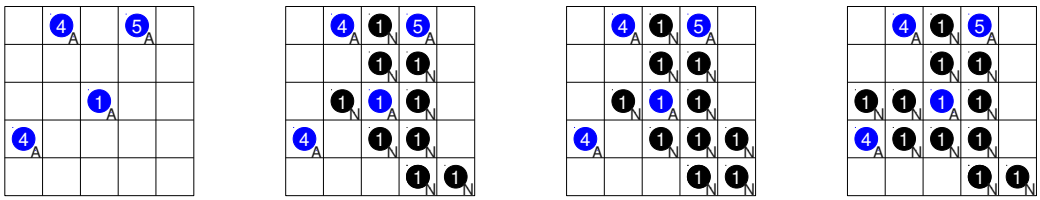
Una celda es *marcable* cuando (a) es parte de una isla (b) está desmarcada, y (c) es lindante con alguna celda marcada. Escribir una función `hayCeldaMarcable` que indique si el tablero tiene alguna celda marcable.

[2]Es decir, contiene celdas blancas o celdas indicadoras.



(a) (b) (c) (d)

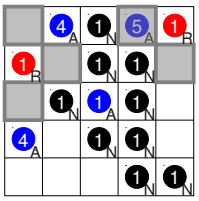
Figura 4: (a) Mapa inicial con cuatro casillas indicadoras. (b) Resolución parcial del mapa inicial con tres islas y un río que no es continuo; notar, además, que una de las islas contiene dos indicadores. (c) Solución del mapa inicial que es incorrecta porque tiene una región de 2×2 celdas negras (remarcadas con gris), y porque la isla que contiene el indicador 5 tiene sólo 4 celdas. (d) Solución del mapa inicial que es correcta porque el río es continuo, cada isla tiene la misma cantidad de celdas que la que indica su única casilla indicadora, y no hay regiones de 2×2 celdas negras.



(a) (b) (c) (d)

Figura 5: Representaciones de los mapas de la Figura 4 en Gobstones.

Sugerencia: escribir una función `esCeldaMarcable` que determine si la celda actual es marcable.
Usar sin definir: `lindanteMarcada(d)` que denota `True` cuando existe una celda en dirección `d` que además está marcada.[3]
Ejemplo: las celdas resaltadas son las **únicas** marcables, porque están desmarcadas, forman parte de una isla y son lindantes (al N, E, S u O) a celdas marcadas.



Ejercicio 21

Escribir un procedimiento `MarcarIsla` que marque todas las celdas de la isla que contiene la celda actual. Para ello, **debe** comenzar marcando la celda actual, y luego **debe** marcar todas las celdas marcables hasta que no queden más celdas marcables.[4]
Precondición: no hay ninguna celda marcada en el tablero antes de la invocación del pro-

[3] Notar que `lindanteMarcada` no tiene precondición. En caso que la celda actual sea la última en dirección `d`, la función denota `False`.

[4] La estrategia propuesta funciona. No hace falta justificar este hecho, basta con implementar lo que se indica.

cedimiento. La celda actual forma parte de una isla.
Usar sin definir: `IrACeldaMarcable` que posiciona el cabezal en una celda marcable. Como precondición, debe haber al menos una celda marcable en el tablero.
Ejemplos: El tablero de la derecha resulta de aplicar `MarcarIsla` en el tablero de la izquierda cuando el cabezal se encuentra en la celda recuadrada.

Recordemos que para resolver un mapa hay que dibujar el río de forma tal que cada isla contenga exactamente una celda indicadora. Más aún, cada isla debe tener tantas celdas como el número que aparece en la celda indicadora (ver Figura 4 (d)). Las siguiente funciones se utilizan para verificar qué islas fueron resueltas.

Ejercicio 22

Escribir la función `tamañoIsla` que denote la cantidad de celdas de la isla que contiene la celda actual.
Precondición: no hay ninguna celda marcada en el tablero antes de invocar la función. La celda actual forma parte de una isla.
Observación: recordar la función `nroBolitasTotal(c)` (ejercicio 16, práctica 5).
Ejemplos: el tamaño de la isla de la izquierda es 10, el de la derecha es 6.

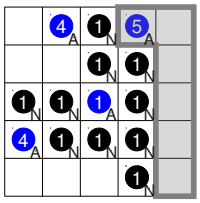
Ejercicio 23

Escribir la función `nroIndicadoras` que denota la cantidad de celdas indicadoras de la isla que contiene la celda actual.
Precondición: no hay ninguna celda marcada en el tablero antes de invocar la función. La celda actual forma parte de una isla.
Ejemplo: la isla de la izquierda tiene 2 indicadores, el de la derecha 1.

Ejercicio 24

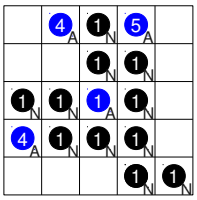
Escribir la función `indicador` que denote el valor del indicador de la isla que contiene la celda

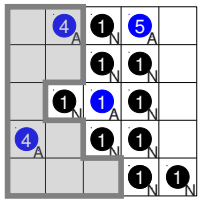
actual.
Precondición: hay una única celda indicadora en la isla. No hay ninguna celda marcada antes de invocar la función. La celda actual forma parte de una isla.
Usar sin definir: `IrAIndicadorMarcado` que posiciona el cabezal en alguna celda indicadora que está marcada. Como **precondición**, tiene que haber al menos una celda indicadora marcada.
Ejemplo: el indicador de la isla marcada es 5.

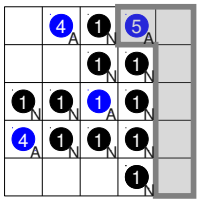


Ejercicio 25

Decimos que una isla esta *resuelta* si (a) contiene exactamente una celda indicadora, y (b) la cantidad de celdas de la isla es igual al número de la celda indicadora. Escribir la función `quedanIslasSinResolver` que retorna true si el mapa contiene alguna isla no resuelta.
Precondición: no hay ninguna celda marcada en el tablero antes de invocar la función.
Ejemplos: en el tablero de la izquierda todas las islas están resueltas; en el tablero del centro la isla marcada no está resuelta dado que contiene dos celdas indicadoras; en el tablero de la derecha la isla marcada no está resuelta porque tiene 6 celdas y su indicador es 5.







Para finalizar, la siguiente función se usará para determinar si el río no tiene regiones pintadas de 2×2 celdas.

Ejercicio 26

Un *lago* es una región de 2×2 celdas del río (ver Figura 4 (c)). Escribir una función `estaEnLago` que indique si la celda actual es parte de un lago.
Usar sin definir: `hayRioAl(d)` que denota `True` si la celda en dirección `d` es parte del río; como precondición, debe ser posible moverse en dirección `d`. `hayRioDiag(d)` que denota `True` si la celda en dirección `d + siguiente(d)` es parte del río; como precondición, debe ser posible moverse tanto en dirección `d` como `siguiented(d)`.
Ejemplo: en el tablero de la izquierda la celda bajo el cabezal —recuadrada en gris— forma parte de un lago, mientras que en el tablero de la derecha la celda bajo el cabezal no forma parte de un lago.

5. Batalla Naval

La *Batalla Naval* es un juego conocido. Cada uno de dos jugadores cuenta con una flota de barcos que colocan sobre un tablero propio sin que el otro sepa dónde. Cada barco puede ocupar una o más celdas contiguas (cada una de ellas representa una "sección" del barco). Luego se turnan para disparar bombas sobre las celdas del tablero del oponente, anunciando en cada caso las coordenadas donde las mismas son arrojadas. Si en la coordenada anunciada hay una sección de un barco del oponente, entonces esa sección es destruida. El barco se declara "hundido" una vez que todas sus secciones son destruidas. Gana el primero que logra hundir todos los barcos de la flota del oponente.

Vamos a modelar este juego en GOBSTONES. Para ello el tablero estará dividido en dos partes, a saber la *parte de datos* y la *parte de juego* (ver Fig. 6).

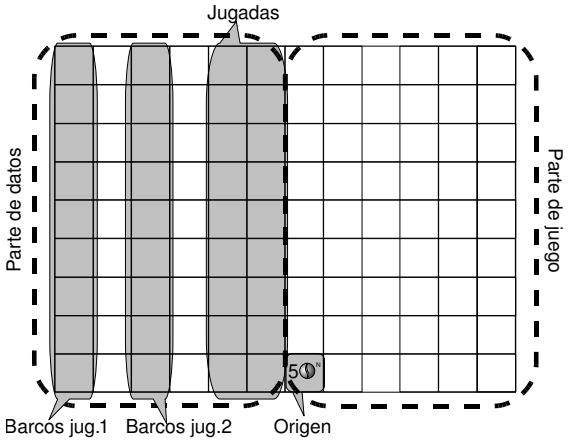


Figura 6: División del tablero

Parte de datos

La parte de datos se usa para codificar los datos del juego y ocupa las primeras seis columnas. Las primeras 4 de estas están destinadas a codificar los barcos de ambos jugadores. La columna 1 codifica los barcos del jugador 1 mientras que la columna 3 los del jugador 2 (la finalidad de las columnas 2 y 4 será explicado en breve). Cada barco se representa a través de tres ingredientes:

- Una coordenada: indicada con bolitas azules (eje de las x) y bolitas verdes (eje de las y)

- Un tamaño: indicado con bolitas rojas
- Una dirección: indicada con bolitas negras (1 = norte, 2 = este, 3 = sur, 4 = oeste)

El resto de la parte de datos (columnas 5 y 6) codifica las jugadas, representadas como coordenadas. En estas columnas, si una celda corresponde con una jugada de un jugador entonces la celda lindante al sur tiene los datos de la jugada del oponente. Cuando hay más jugadas que el alto del tablero, estas continúan al tope de la columna 6.

Todas las coordenadas se miden con respecto al origen (esquina Sur-Oeste) de la parte de juego. La misma está indicada con 5 bolitas negras.

Parte de juego

En la parte de juego se dibujarán los barcos y se jugará. Las bolitas azules marcarán los barcos del jugador 1 y las verdes los del jugador 2. Con bolitas rojas se demarcarán las secciones de barcos que han sido destruidas. Por limitaciones de espacio las flotas de ambos jugadores se colocan sobre la parte de juego (se asume que los barcos no se solapan).

Problemas

Ejercicio 27

Asuma que el cabezal se encuentra en una celda de la parte de datos que codifica un barco. Escribir un procedimiento `RenderBarco` que tome un color por parámetro y "dibuje" ese barco con el color dado en la parte de juego del tablero. Debe tenerse en cuenta que:

1. La primera sección del barco se marca con dos bolitas del color dado
2. La primera sección del barco también debe incluir las bolitas negras que indican la dirección
3. Las demás secciones del barco solamente contendrán una bolita del color dado

A modo de ejemplo, la figura 7 muestra cómo se dibujan los barcos sobre la parte de juego basándose en la codificación de la parte de datos. Se exhibe cómo quedaría el tablero al finalizar de dibujar todos los barcos de la parte de datos asumiendo que las celdas enumeradas del 1 al 6 contienen:

Celda	Azul (x)	Verde (y)	Negro (dir)	Rojo (tamaño)
1	0	2	1	3
2	2	4	3	2
3	5	8	3	5
4	1	0	2	3
5	4	2	4	2
6	1	6	2	4

Nota: Puede asumir la existencia de un procedimiento `IrACoordenadaBN` que toma dos números y ubica el cabezal en esa coordenada **relativa** al origen de la parte de juego. Ej. `IrACoordenadaBN(0,0)` se ubica sobre la celda que tiene 5 bolitas negras.

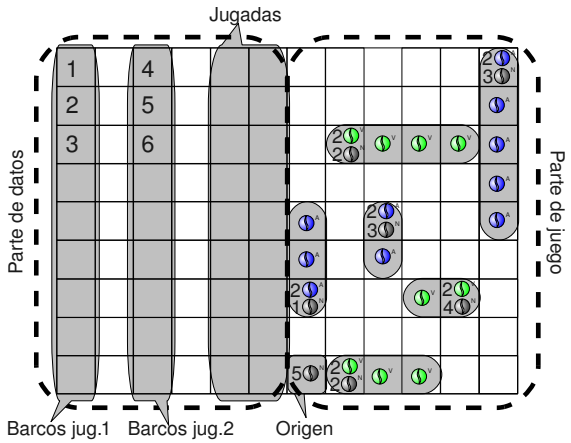


Figura 7: Dibujo de barcos en base a su codificación

Ejercicio 28

Escribir un procedimiento `RenderBarcos` que tome un color por parámetro y recorre todos los barcos del jugador de ese color y los "dibuje" en la parte de juego utilizando el procedimiento anterior. Utilice las columnas libres de la parte de datos para demarcar (con seis bolitas negras) el barco que está siendo procesado actualmente. De esta manera, por cada barco que se encuentre codificado se deberá:

- poner seis bolitas negras en la celda lindante al Este
- dibujar el barco utilizando el procedimiento anterior,
- retornar a la celda inicial, buscando las seis bolitas negras,
- eliminar las seis bolitas negras.

Nota: Estructurarlo como recorrido. Puede asumir que el cabezal se encuentra en el tope de la columna 1 o 3 según el color sea azul o verde.

Ejercicio 29

Escribir una función `estaHundido` que, asumiendo que el cabezal se encuentra en la celda de la parte de datos que codifica un barco, retorne un valor de verdad indicando si el mismo está hundido o no. Recordar que las secciones destruidas se indican con una bolita roja, y un barco se considera hundido si todas sus secciones han sido destruidas.

Ejercicio 30

Escribir una función `perdio`, que reciba por parámetro un color (azul o verde) y determine si todos los barcos de ese color han sido hundidos. **Nota:** Estructurar como recorrido sobre las celdas de la parte de datos del tablero y apelar al ejercicio anterior.

Ejercicio 31

Escribir un procedimiento `Disparo` que reciba un color y simule un disparo sobre los barcos

de ese color. La coordenada del disparo deberá tomarla de la celda actual. En caso de ser efectivo el disparo colocar una bolita roja en el lugar del impacto, indicando que se destruye esa sección de barco.

Nuevamente, se pide que utilice seis bolitas negras para demarcar el disparo actual que está siendo procesado.

Ejercicio 32

Escribir una funcion *simularBN* que, partiendo de un tablero en el cual la parte de juego se encuentra en blanco, utilice la información contenida en la parte de datos para simular el juego y devuelva el color ganador.

Naturalmente deberá comenzar renderizando todos los barcos codificados en las columnas 1 y 3, para luego recorrer uno a uno los disparos.

La simulación termina cuando uno de los jugadores gana o bien cuando se acaban los disparos codificados (la celda que correspondería al siguiente disparo se encuentra en blanco o se alcanza el límite sur de la columna 6).

La función debe retornar el color de los barcos del jugador ganador, o bien el color negro para indicar que con esa secuencia de disparos ninguno de los dos jugadores llegó a hundir todos los barcos del contrincante.