Universidad Simón Bolívar
Dpto. de Computación y Tecnología de la Información
CI3661 - Taller de Lenguajes de Programación I
Enero-Marzo 2016

Programación Orientada a Objetos - Tarea

1. Juegos de Manos

En este ejercicio de programación, usaremos el popular juego "Piedra, Papel o Tijeras", de manera que Ud. pueda aplicar los conceptos de despacho doble y posiblemente *duck-typing* discutidos en clase. Con este propósito, modelaremos el juego con una estructura específica usando Ruby

1.1. Los Movimientos (1 punto)

Utilizaremos la clase `Movement` para representar la noción de movimiento ejecutado por un jugador, siendo necesario contar con las subclases `Rock`, `Paper` y `Scissors`, para representar los movimientos específicos. Es necesario implantar los métodos:

- `to_s` para mostrar el invocante como un `String`.
- `score(m)` que determine el resultado de la jugada entre el invocante y el movimiento `m`, correspondiente al contrincante, que es recibido como argumento. El resultado de `score` debe ser una tupla que representa la ganancia en puntos resultado de la jugada: el primer elemento de la tupla representa la ganancia del invocante, mientras que el segundo elemento representa la ganancia del contrincante. Así, la tupla resultante debe ser `[1,0]`, `[0,1]` o `[0,0]` dependiendo de los movimientos involucrados.

1.2. Las Estrategias (3 puntos)

Cada jugador será representado por un objeto de la clase `Strategy`. La clase `Strategy` permite generar el siguiente movimiento del jugador, quizás aprovechando las jugadas anteriores propias, del rival, o ambas, como base de referencia. Toda estrategia debe proveer los métodos:

- `next(m)` que genera el próximo `Movement` usando como información adicional, si le conviene, el `Movement m` suministrado como argumento. Note que este método debe retornar un objeto en *alguna* de las clases `Rock`, `Paper` o `Scissors` – es un error retornar un `String`.
- `to_s` para retornar el invocante como `String`. Siempre debe mostrar el nombre simbólico, pero debe estar especializada para mostrar los parámetros de configuración específicos de cada estrategia discutida más abajo.
- `reset` para llevar la estrategia a su estado inicial, cuando esto tenga sentido.

Usted debe implantar al menos las siguientes especializaciones de `Strategy`:

- `Uniform`, construida recibiendo una lista de movimientos posibles y seleccionando cada movimiento usando una distribución uniforme sobre los movimientos posibles, i.e.

  ```
  r = Uniform.new( [ :Rock, :Scissors, :Paper ] )
  ```

  Al construir una instancia de esta estrategia, es necesario eliminar duplicados y verificar que haya al menos una estrategia en la lista, en caso contrario emitir una excepción describiendo el error.

- `Biased`, construida recibiendo un mapa de movimientos posibles y sus probabilidades asociadas, de modo que cada movimiento use una distribución sesgada de esa forma. Al construir una instancia de esta estrategia, es necesario eliminar duplicados y verificar que haya al menos una estrategia en el mapa, en caso contrario emitir una excepción describiendo el error. Las probabilidades asociadas a cada tipo de movimiento serán números enteros, i.e.

  ```
  b = Biased.new( { :Rock => 1, :Scissors => 3, :Paper => 2 } )
  ```

  Resultando en probabilidades 1/6, 1/2 y 1/3 respectivamente.

- `Mirror`, cuya primera jugada es definida al construirse, pero a partir de la segunda ronda siempre jugará lo mismo que jugó el contrincante en la ronda *anterior*.

- `Smart`, cuya jugada depende de analizar las frecuencias de las jugadas hechas por el oponente hasta ahora. La estrategia debe recordar las jugadas previas del oponente, y luego decidir de la siguiente forma:

  - Sean p, r y s la cantidad de veces que el oponente ha jugado `Paper`, `Rock` o `Scissors`, respectivamente.
  - Se genera un número entero al azar n entre 0 y $p + r + s - 1$.
  - Se jugará `Scissors` si $n \in [0, p)$, `Paper` si $n \in [p, p+r)$ o `Rock` si $n \in [p+r, p+r+s)$

Notará que varias de las estrategias requieren el uso de números al azar. La librería `Random` provista por Ruby tiene toda la infraestructura necesaria. Con el propósito de poder evaluar de manera semiautomática sus soluciones, es necesario que todos los generadores de números al azar tengan exactamente la misma semilla, para ello declare una constante de clase en `Strategy` con el valor 42, a ser utilizada cada vez que necesite una semilla.

1.3. El Juego (2 puntos)

Un juego ocurre entre dos jugadores, cada uno definido por su estrategia. La clase `Match` debe construirse recibiendo un mapa con los nombres y estrategias de los jugadores

```
m = Match.new( { :Deepthought => s1, :Multivac => s2 } )
```

donde `s1` y `s2` son instancias de alguna de las estrategias. Es necesario verificar que hay exactamente dos jugadores y que en efecto se trata de estrategias, generando excepciones descriptivas del problema cuando esas condiciones no se cumplan.

Tendremos interés en observar el desarrollo del juego de varias maneras:

- `rounds(n)`, con `n` un entero positivo, debe completar `n` rondas en el juego y producir un mapa indicando los puntos obtenidos por cada jugador y la cantidad de rondas jugadas.
- `upto(n)`, con `n` un entero positivo, debe completar tantas rondas como sea necesario hasta que alguno de los jugadores alcance `n` puntos, produciendo un mapa indicando los puntos obtenidos por cada jugador y la cantidad de rondas jugadas.
- `restart`, debe llevar el juego a su estado inicial.

Ha de ser posible continuar un juego en curso, i.e. si se ejecutara

```
m.rounds(10)    # Se ejecutan 10 rondas
m.rounds(20)    # Se ejecutan 20 rondas adicionales
r = m.upto(100) # Se ejecutan las rondas hasta que alguno lle-
gue a 100
{ :Multivac => 84, :Deepthought => 100, :Rounds => 238 }
```

el juego produciría los resultados de las primeras 10 rondas, a los cuales *acumularía* los resultados de las siguientes 20 rondas y por último continuaría hasta que algún jugador acumule 100 puntos.

2. Búsqueda generalizada

2.1. Arboles y grafos explícitos (2 puntos)

Considere la siguiente definición para una clase que representa árboles binarios

```
class BinTree
  attr_accessor :value,  # Valor almacenado en el nodo
                :left,   # BinTree izquierdo
                :right   # BinTree derecho
  def initialize(v,l,r)
    # Su código aquí
  end
  def each(b)
    # Su código aquí
  end
end
```

en la cual el propósito de `each` es recibir un *bloque* que será utilizado para iterar sobre los hijos del nodo, cuando estén definidos.

Ahora, considere la siguiente definición para una clase que representa grafos arbitrarios a partir de un nodo específico, indicando sus sucesores

```
class GraphNode
  attr_accessor :value,   # Valor alamacenado en el nodo
                :children # Arreglo de sucesores GraphNode
  def initialize(v,c)
    # Su código aquí
```

```
  end
  def each(b)
    # Su código aquí
  end
end
```

en la cual el propósito de `each` es recibir un *bloque* que será utilizado para iterar sobre los hijos del nodo, cuando estén definidos.

2.2. Recorrido BFS es un comportamiento (4 puntos)

Se desea que Ud. implante una infraestructura de búsqueda BFS[1] que pueda ser utilizada por *ambas* clases, empleando la técnica de *mixins* estudiada en clase. Se espera que su *mixin* ofrezca la siguiente interfaz de programación

- `find(start,predicate)` que comienza la búsqueda BFS a partir del objeto `start` hasta encontrar el primer objeto que cumpla con el predicado `predicate`, y lo retorna. Si se agota la búsqueda sin encontrar objetos que cumplan el predicado, se retorna el objeto indefinido.
- `path(start,predicate)` que comienza la búsqueda BFS a partir del objeto `start` hasta encontrar el primer objeto que cumpla con el predicado `predicate`, y retorna el *camino* desde `start` hasta el nodo encontrado, en forma de `Array` de objetos Si se agota la búsqueda sin encontrar objetos que cumplan el predicado, se retorna el objeto indefinido.
- `walk(start,action)` que comienza un recorrido BFS a partir del objeto `start` hasta agotar todo el espacio de búsqueda, ejecutando el cuerpo de código action sobre cada nodo visitado y retornando un `Array` con los nodos visitados. Si el cuerpo de código `action` se omite, sólo debe retornar el `Array` con los nodos visitados.

Para la implantación de su *mixin* sólo puede asumir que las clases usuarias disponen del método de acceso `value` para obtener el valor almacenado en un nodo, y del método `each` para recorrer todos los hijos de un nodo particular. Los métodos de su *mixin* **no pueden crear variables de instancia ni de clase adicionales**. Seguramente necesitará métodos adicionales dentro de su *mixin* para asistirlo en la implantación de su recorrido BFS.

2.3. Arboles implícitos (3 puntos)

Una técnica para resolver problemas de configuración, sean juegos o acertijos, consiste en explorar un árbol implícito de expansión, incluyendo recortes inteligentes para reducir la complejidad en tiempo y espacio. La técnica comienza por definir un estado inicial, y luego generar implícitamente los estados siguientes como nodos hijos. Después de generar los hijos válidos (sujeto a las restricciones particulares del problema), se verifica si alguno de ellos el estado final deseado, de lo contrario, se repite el procedimiento. Un problema simple que entretiene a chicos y grandes es el de "Lobo, Cabra y Repollo" [3], y es el problema que queremos modelar.

Suponga la siguiente definición de clase que permite modelar un *estado* del problema

```
def LCR
  attr_reader :value
  def initialize(?)  # Indique los argumentos
    # Su código aquí
  end
  def each(p)
    # Su código aquí
  end
  def solve
    # Su código aquí
  end
end
```

en la cual:

- El atributo `value`, que sólo se puede leer, corresponde a un estado particular del problema representado como un `Hash`. El `Hash` tiene tres claves `where`, `left` y `right`, para indicar la posición del bote con un valor del tipo `Symbol`, así como los contenidos de las orillas izquierda y derecha usando listas de `Symbol`.
- El método `each` recibe un bloque que será utilizado para iterar sobre los hijos del estado actual. Los hijos deben ser generados dinámicamente, y el cuerpo de código sólo debe iterar sobre aquellos que tengan sentido en el contexto del problema de búsqueda.
- El método solve resuelve el problema de búsqueda

  ```
  initial = LCR.new(...)
  initial.solve()
  ... salen en pantalla los movimientos necesarios...
  ```

La clase LCR debe aprovechar el *mixin* de BFS para resolver el problema.

3. Entrega de la Implementación

Ud. debe entregar un archivo `.tar.gz` o `.tar.bz2` (**no** puede ser `.rar`, ni `.zip`) que al expandirse genere un directorio con el nombre de su grupo (e.g. `G42`) dentro del cual encontrar **solamente**:

- Los archivos `rps.rb` y `bfs.rb` conteniendo el código fuente Ruby para implantar las clases y mixins correspondientes a cada sección.
- Se evaluará el correcto estilo de programación empleando una indentación adecuada y consistente en *cualquier* editor (”profe, en el mío se ve bien” es inaceptable).
- El archivo **debe** estar completa y correctamente documentado[2]. Es **inaceptable** que la documentación tenga errores ortográficos.
- Su programa será corregido usando Ruby 2.1 empleando exclusivamente las librerías estándar incluidas con la distribución. No está permitido utilizar librerías externas ni gemas.

- Puede escribir métodos adicionales si los necesita, pero estos no pueden ser públicos.
- **Fecha de Entrega.** Martes 2016-03-11 hasta las 23:59 VET
- **Valor de Evaluación.** Quince (15) puntos.

4. Referencias

Referencias

[1] BFS en Wikipedia

[2] Ruby Documentation System

[3] Fox, Goose and Bag of Beans en Wikipedia