Justificación de Métodos

Edgar Aguadé Nadal

Mayo 2024

`void City::trade(City &visitor_city, const ProductSet &product_set);`
Al final del documento se encuentra el código completo.

Precondición Cierto.

Postcondición Las ciudades han comerciado entre sí todos aquellos productos posibles. O de un modo más formal:

Sean:

- $city_{local}$:= Parámetro implícito
- $id(product)$:= El identificador del producto "product"
- $exc(product)$:= Exceso del producto "product"

$\forall product_{local} \in city_{local} \forall product_{visitor} \in city_{visitor}$ tales que:
$id(product_{local}) = id(product_{visitor}) \land exc(product_{local}) * exc(product_{visitor}) < 0$

Se cumple que, siendo $t := min(|exc(product_{local})|, |exc(product_{visitor})|)$
$exc(product_{local}) > 0 \implies own_{local} = own_{local} - t \land own_{visitor} = own_{visitor} + t$
$exc(product_{local}) < 0 \implies own_{local} = own_{local} + t \land own_{visitor} = own_{visitor} - t$

O lo que viene a ser lo mismo, todos aquellos productos donde una ciudad tenga más de las que necesita, la otra menos de las que necesita y viceversa, han sido comerciados. La cantidad comerciada "t" es el mínimo de los excedentes, pues no puede ni haber superávids ni que una ciudad que tiene exceso quede con menos unidades de las que necesita.

Invariante

- $city_{local}.begin() \leq product_{local} \leq city_{local}.end()$
- $city_{visitor}.begin() \leq product_{visitor} \leq city_{visitor}.end()$

- $\forall product_i < product_{local} \in city_{local} \forall product_j < product_{visitor} \in city_{visitor}$ $city_{local}, city_{visitor}$ han intercambiado dichos productos cumpliendo la Post-condición.

1 Justificación

1.1 Inicializaciones

Inicialmente no se ha comerciado ningún producto, por lo que inicializamos ambos iteradores a al inicio de sus respectivos inventarios: $product_{local} = city_{local}.begin() \wedge product_{visitor} = city_{visitor}.begin()$, cumpliendo así las dos partes del *Invariante*, por un lado que ambos iteradores se encuentran entre el principio y el final, y, por otro lado, que todos los productos anteriores a los iteradores han sido comerciados.

1.2 Codición de salida

Se puede salir del bucle por dos razones:

- $product_{local} = city_{local}.end()$

Lo cual indicaria, por el invariante, que hemos explorado todos los productos de $city_{local}$ y, en consecuencia, todos los de $city_{visitor}$ tales que $id(product_j) \leq id(product_{local})$.

Por lo tanto, $\forall product_j$ tal que $id(product_j) > id(product_{local})$ $\nexists product_i$ tal que $id(product_i) = id(product_j)$

De lo contario significaría que $\exists product_j$ tal que $id(product_j) > product_{local} \wedge \exists product_i$ tal que $id(product_i) = id(product_j)$. Por el invariante obtenemos que $product_i \geq city_{local}.end()$ lo cual es absurdo pues no hay más productos después del final del inventario.

- $product_{visior} = city_{visitor}.end()$

Completamente análogo a lo acabado de demostrar.

1.3 Cuerpo del bucle

Para analizar correctamente el cuerpo del bucle diferenciaremos claramente en tres casos:

- $id(product_{local}) < id(product_{visitor})$

Por el *Invariante* todos los productos anteriores a $product_{local}, product_{visitor}$ se han intercambiado si cumplian con $id(product_{local}) = id(product_{visitor})$. En estar ordenados de menor a mayor índice deducimos que $\nexists product_j \in city_{visitor}$ tal que $id(product_j) < id(product_{visitor}) \wedge id(product_{local}) = id(product_j)$ De lo contrario deduciríamos que $\exists product_j \in city_{visitor}$ tal que $id(product_j) < id(product_{visitor})$ que no ha comerciado, absurdo por el *Invariante*. De este modo, avanzamos/descartamos $product_{local}$ puesto que estamos seguros que ese producto no va a ser comerciado.

- $id(product_{local}) > id(product_{visitor})$

$id(product_{local}) > id(product_{visitor}) \iff id(product_{visitor}) < id(product_{local})$ y por lo tanto, análogo al caso anterior.

- $id(product_{local}) = id(product_{visitor})$

Para que se vuelva a cumplir el *Invariante*, las ciudades deben comerciar si se cumple $exc(product_{local}) * exc(product_{visitor}) < 0$. Con este objetivo precalculamos "t" la cantidad transaccionada. del modo que:

$$t := min(|exc(product_{local})|, |exc(product_{visitor})|)$$

De este modo podemos ver que actuamos según dos casos:

$exc(product_{local}) > 0 \implies exc(product_{visitor}) < 0$ por lo que $city_{local}$ vende y $city_{visitor}$ compra: $own_{local} = own_{local} - t \wedge own_{visitor} = own_{visitor} + t$

Por otro lado...

$exc(product_{local}) < 0 \implies exc(product_{visitor}) > 0$ por lo que $city_{local}$ compra y $city_{visitor}$ vende: $own_{local} = own_{local} + t \wedge own_{visitor} = own_{visitor} - t$

A su vez, en cada paso actualizamos los pesos y volúmenes correspondientes de las ciudades.

Por último, avanzamos $product_{local}, product_{visitor}$ para cumplir con el *Invariante*.

1.4 Finalización

Como podemos apreciar, en cada iteración o avanzamos $product_{local}$ o avanzamos $product_{visitor}$ o ambos simultáneamente. Por lo que la distancia entre estos y $city_{local}.end(), city_{visitor}.end()$ respectivamente, disminuye en cada iteración, de lo que deducimos que eventualmente el bucle finalizará.

```
void River::findOptimalRoute(BinTree<string> structure, Travel
        current_travel, Travel &best_travel) const;
```
Al final del documento se encuentra el código completo.

Precondición $travel_{current}$ contiene el estado del viaje realizado para llegar a la ciudad actual.

Postcondición $travel_{best}$ guarda el mejor viaje realizado hasta el momento y $travel_{current}$ contiene el estado del viaje realizado una vez pasada a la ciudad actual.

2 Justificación

2.1 Caso Base

2.1.1 $structure.empty()$

$structure.empty() \iff \nexists structure.left(), structure.right() \iff$ No hay más nodos que explorar y por lo tanto se ha acabado el camino.

En este caso no se actualiza $travel_{best}$ ni $travel_{current}$ por lo que sigue cumpliendo la *Postcondición*.

2.1.2 $\neg structure.empty()$

$\neg structure.empty() \implies structure.value()$ es una ciudad de $travel_{current} \implies$ la longitud del viaje aumenta $\wedge$ se debe intentar comerciar con dicha ciudad.

En haber cambiado el estado de $travel_{current} \implies$ hay que asegurar que $travel_{best}$ sigue siendo el mejor viaje por lo que comparamos ambos viajes y en caso que $travel_{current}$ sea mejor lo remplazamos. Así pues, se cumple la *Postcondición*.

2.2 Caso Inductivo

Hipótesis El árbol no es vacio. Es decir, $\neg structure.empty()$

Aislando el caso directo en que $travel_{current}.objectiveAchived()$ *es cierto (el barco ya ha vendido y comprado todo lo posible y por lo tanto no debe continuar la ruta), pues en este caso* $\neg structure.empty() \implies$ *Postcondición. Veamos entonces el caso inductivo que es el interesante:*

$\neg structure.empty() \implies \exists structure.left() \wedge structure.right()$ Por lo que podemos ver dos casos

- $structure.left().empty() \wedge structure.right().empty()$

  En este caso no podemos continuar el camino, puesto que ya hemos llegado al final del árbol. Por otro lado, en ser $\neg structure.empty() \implies$ *Postcondición.*

- $\neg structure.left().empty() \wedge \neg structure.right().empty()$

  $\implies \exists structure.left().value(), structure.right().value()$ Por lo tanto, como se cumple la *Precondición* se cumplirá la *Postcondición.*

Decrecimiento

$structure.left(), structure.right() \subseteq structure$ por lo que las llamadas se hacen cada vez con un árbol más pequeño, asegurando así que eventualmente se llega a una hoja/final.

Código Método Comerciar

```
1   void City::trade(City &visitor_city, const ProductSet &product_set)
        {
2       map<int, ProductInventoryStats>::iterator visitor_product =
            visitor_city.inventory.begin();
3       map<int, ProductInventoryStats>::iterator local_product =
            inventory.begin();
4
5       // Vamos a iterar los dos inventarios a la vez de arriba a abajo
            , aprovechando que estos estan
6       // ordenados de menor a mayor indice, podremos ir avanzando los
            dos "montones" individualmentel
7       // e ir encontrando todos los indices que coincidan.
8       while (local_product != inventory.end() and visitor_product !=
            visitor_city.inventory.end()) {
9           // Guardamos los identificadores de cada producto en una
                variable para mayor legibilidad.
10          int local_id = local_product->first, visitor_id =
                visitor_product->first;
11
12          if (local_id == visitor_id) { // Las dos ciudades tienen el
                mismo producto ==> pueden comerciar.
13              int local_owned = local_product->second.getOwned();
14              int visitor_owned = visitor_product->second.getOwned();
15              int local_excess = local_product->second.getOwned() -
                    local_product->second.getNeeded();
16              int visitor_excess = visitor_product->second.getOwned() -
                    visitor_product->second.getNeeded();
17
18              if (local_excess*visitor_excess < 0) {
19              // local_excess*visitor_excess < 0 ==> (local_excess > 0 y
                     visitor_excess < 0) o (local_excess < 0 y
                    visitor_excess > 0)
20
21                  // Se transacciona el excedente mas pequeno, ya que si
                        la visitante necesita mas de lo que
22                  // la local tiene, esta le dara todo su excedente. Por
                        otro lado, si el excedente de la local
23                  // es mayor que lo que le falta a la visitante para
                        llegar a lo que necesita, la local le dara
24                  // solamente esta cantidad.
25                  int transacted = min(abs(local_excess), abs(
                        visitor_excess));
26
27                  // Se debe recalcular el peso y volumen total de ambas
                        ciudades.
28                  int product_weigth = product_set.getWeightById(local_id
                        );
29                  int product_volume = product_set.getVolumeById(local_id
                        );
30                  // Precalculamos la variacion de pesos y volumenes.
31                  int weight_variance = transacted*product_weigth;
```

```
32                  int volume_variance = transacted*product_volume;
33
34                  int local_new_owned, visitor_new_owned;
35                  if (local_excess > 0) { // ==> visitor_exces < 0 ==> p.
                        i vende "transacted" visitor compra "transacted".
36                      local_new_owned = local_owned - transacted;
37                      decreaseTotalWeightAndVolumen(weight_variance,
                            volume_variance);
38                      visitor_new_owned = visitor_owned + transacted;
39                      visitor_city.increaseTotalWeightAndVolumen(
                            weight_variance, volume_variance);
40                  } else { // local_excess < 0 ==> visitor_exces > 0 ==>
                        p.i compra "transacted" visitor vende "transacted".
41                      local_new_owned = local_owned + transacted;
42                      increaseTotalWeightAndVolumen(weight_variance,
                            volume_variance);
43                      visitor_new_owned = visitor_owned - transacted;
44                      visitor_city.decreaseTotalWeightAndVolumen(
                            weight_variance, volume_variance);
45                  }
46
47                  local_product->second.setOwned(local_new_owned);
48                  visitor_product->second.setOwned(visitor_new_owned);
49              }
50              ++local_product;
51              ++visitor_product;
52          }
53          else if (local_id < visitor_id) ++local_product;
54          else ++visitor_product; // (local_id > visitor_id) == (
                visitor_id < local_id)
55      }
56  }
```

Código del Método para Encontrar la ruta mas provechosa

```
1    void River::findOptimalRoute(BinTree<string> structure, Travel
         current_travel, Travel &best_travel) const {
2        if (not structure.empty()) {
3            // La ruta del viaje tiene una parada mas ==> Aumenta la
                 longitud del camino.
4            current_travel.increaseLength();
5            // Simulamos la transaccion
6            tryTransaction(structure.value(), current_travel);
7            // La transaccion puede hacer que el viaje actual sea mejor
                 que el mejor viaje hasta la fecha
8            // para seguir cumpliendo la Postcondicion comprovamos y
                 actualizamos.
9            if (current_travel.betterTravelThan(best_travel)) best_travel
                  = current_travel;
10
11           if (not (current_travel.objectiveAchived() or structure.left
                 ().empty())) {
12               // Travel::bestTravelThan en caso de empate total
                     considera el paramentro implicito como mejor,
13               // Por lo que para cumplir la condicion de que en caso de
                     empate total se quede con el viaje
14               // mas cercano a la izquierda, debemos visitar la derecha
                     antes que la izquierda de este modo
15               // "current_travel" siempre estara mas cercano a la
                     izquierda que a la derecha.
16
17               findOptimalRoute(structure.right(), current_travel,
                     best_travel);
18               findOptimalRoute(structure.left(), current_travel,
                     best_travel);
19           }
20       }
21       // Caso base: structure.empty() ==> Se ha llegado al final de la
              ruta ==>
22       // ==> No se actualiza nada, se acabam las llamadas recursivas.
23   }
```