Mohamed Amara El Houiti

Justificación de la función Comerciar

Invariante del Bucle

Al inicio y al final de cada iteración del bucle **while**, todos los productos de las dos ciudades que comercian han sido visitados hasta donde apuntan los iteradores it y it2, y se han comerciado estos productos. Por lo tanto, los iteradores it y it2 siempre apuntan al siguiente producto a visitar o al final del mapa de productos.

Esto significa que en cualquier punto durante la ejecución del bucle **while**, los productos anteriores a los apuntados por it y it2 han sido visitados y se han realizado los comercios necesarios.

Inicialización

Después de que se ejecute **it = productos.begin()** y **it2 = ciudad2.productos.begin()**, se debe cumplir el invariante. Podemos ver que si:

- En este punto, **it y it2** apuntan al primer producto en los mapas de productos de ciudad1 y **ciudad2.** No se ha realizado ningún comercio todavía. El invariante es verdadero ya que no hay productosantes de los primeros elementos y no se ha realizado ningún comercio, por lo que se cumplen las condiciones de que todos los productos hasta donde apuntan **it y it2** (es decir, ninguno en este momento) han sido procesados.

Condición de Salida

El bucle termina cuando it llega al final del mapa de productos de ciudad1 o it2 llega al final del mapa de productos de ciudad2, **(it == productos.end() o it2 == ciudad2.productos.end())**. En ese momento, todos los productos han sido visitados y procesados según el invariante**.**

Cuerpo del Bucle

Dentro del bucle while nos encontramos con los siguiente pasos:

1. **Comparación de IDs de Productos:**

```
if (it->first < it2->first) {
   ++it;
} else if (it->first > it2->first) {
   ++it2;
} else {
```

- Si el ID del producto en ciudad1 **(it->first)** es menor que el ID del producto en ciudad2 (it2->first), incrementamos **it (++it)**.

- Si el ID del producto en ciudad1 es mayor, incrementamos **it2 (++it2)**.

- Si los IDs son iguales, significa que ambos productos están presentes en ambas ciudades y se puede realizar el comercio.

2. **Comercio de Productos:**

- **Caso 1: Hay sobrante en ciudad1 y ciudad2 necesita el producto**
  Si hay un sobrante en ciudad1 y ciudad2 necesita el producto, se transfiere la cantidad mínima entre lo que sobra en ciudad1 y lo que necesita ciudad2.

- **Caso 2: Hay sobrante en ciudad2 y ciudad1 necesita el producto**
  Si hay un sobrante en ciudad2 y ciudad1 necesita el producto, se transfiere la cantidad mínima entre lo que sobra en ciudad2 y lo que necesita ciudad1.
  En ambos casos, los inventarios de productos y los totales se actualizan adecuadamente.

3. **Incremento de Iteradores:**
   ++it;
   ++it2;
   Después de procesar productos con IDs iguales, ambos iteradores se incrementan para pasar al siguiente producto en cada ciudad.

¿Se mantiene el invariante?

Al inicio de cada iteración, el invariante es verdadero porque todos los productos hasta donde apuntan it y it2 han sido procesados. Durante la iteración, los productos en las posiciones actuales de it y it2 se comparan y se comercian si es necesario. Finalmente, al incrementar los iteradores **(++it o ++it2 o ambos)**, el invariante se mantiene porque ahora ambos apuntan al siguiente producto a procesar o al final del mapa de productos.

Acabamiento

Cada iteración del bucle garantiza que al menos uno de los iteradores **(it o it2)** se incrementa. Dado que los iteradores solo pueden incrementarse hasta que lleguen a **end()**, uno de ellos alcanzará el final del mapa de productos, asegurando que el bucle terminará.

Mohamed Amara El Houiti

Código de la función:

```
void Ciudad::comerciarcon(Ciudad& ciudad2, Cjt_Productos& productos) {
    map<int, pair<int, int>>::iterator it = tiene_neces.begin();
    map<int, pair<int, int>>::iterator it2 = ciudad2.tiene_neces.begin();

    while (it != tiene_neces.end() and it2 != ciudad2.tiene_neces.end()) {
        int idproducto1 = it->first;
        int idproducto2 = it2->first;

        if (idproducto1 < idproducto2) {
            ++it;
        } else if (idproducto1 > idproducto2) {
            ++it2;
        } else {
            int cantTieneciudad1 = it->second.first;
            int cantTieneciudad2 = it2->second.first;

            int sobranteciudad1 = cantTieneciudad1 - it->second.second;
            int sobranteciudad2 = cantTieneciudad2 - it2->second.second;
            int intercambio = 0;

            if (sobranteciudad1 > 0 and cantTieneciudad2 < it2->second.second) {
                if (sobranteciudad1 > it2->second.second - cantTieneciudad2) intercambio = it2->second.second - cantTieneciudad2;
                    else intercambio = sobranteciudad1;
                if (intercambio > 0) {
                    modificartotales(idproducto1, cantTieneciudad1 - intercambio, productos.getProducto(idproducto1).getPeso(), productos.getProducto(idproducto1).getVolumen());
                    ciudad2.modificartotales(idproducto1, cantTieneciudad2 + intercambio, productos.getProducto(idproducto1).getPeso(), productos.getProducto(idproducto1).getVolumen());
                    it->second.first -= intercambio;
                    it2->second.first += intercambio;
                }
            }

            if (sobranteciudad2 > 0 and cantTieneciudad1 < it->second.second) {
                 if (sobranteciudad2 > it->second.second - cantTieneciudad1) intercambio = it->second.second - cantTieneciudad1;
                    else intercambio = sobranteciudad2;
                if (intercambio > 0) {
                    ciudad2.modificartotales(idproducto1, cantTieneciudad2 - intercambio, productos.getProducto(idproducto1).getPeso(), productos.getProducto(idproducto1).getVolumen());
                    modificartotales(idproducto1, cantTieneciudad1 + intercambio, productos.getProducto(idproducto1).getPeso(), productos.getProducto(idproducto1).getVolumen());
                    it->second.first += intercambio;
                    it2->second.first -= intercambio;
                }
            }
            ++it;
            ++it2;
        }
    }
}
```

Mohamed Amara El Houiti

Justificación de la función hacer_viaje

Hipótesis de Inducción:

Asumimos que para cualquier nodo del árbol a, la función **best_ruta** calcula correctamente la mejor ruta a través de los subárboles izquierdo y derecho del árbol que guarda las ciudades Esto significa que, si la función **best_ruta** funciona correctamente para los subárboles de a, entonces también funcionará correctamente para el nodo **a** y para sus subárboles.

Casos Base, árbol vacío:

Si **a.empty()** es true, no hacemos nada ya que no hay ciudades en esta ruta. Esto es correcto porque un árbol vacío (no ciudad) no aporta a la ruta.

Casos Recursivos:

Para cada nodo del árbol ciudades no vacío:

1. **Verificamos y actualizamos inventarios:**
    - **Compra de Productos:**
      Si la ciudad actual **(a.value())** tiene el producto **idcomprar** y hay sobrante, el barco compra la cantidad mínima entre lo que necesita comprar (comprar) y el sobrante en la ciudad.
    - **Venta de Productos**
      Si la ciudad actual **(a.value())** tiene el producto **idvender** y hay sobrante, el barco vende la cantidad mínima entre lo que puede vender (vender) y el sobrante en la ciudad.
2. **Actualizamos la ruta y las cantidades acumuladas:**
    - Añadimos la ciudad actual a **ruta_actual**.
    - Calculamos el total de productos comprados y vendidos en la ciudad actual y lo añadimos a lo q está acumulado.

3. **Comparación y Actualización de la Mejor Ruta:**
    - Si la nueva cantidad acumulada (**totalProductosComprados + totalProductosVendidos**) es mayor que bestcantidadprods, actualizamos mejor_ruta, **bestcantidadprods y mejorLongitud**.
    - Si la nueva cantidad acumulada es igual a **bestcantidadprods**, actualizamos solo si la longitud de la ruta actual (**ruta_actual.size()**) es menor que **mejorLongitud**.

4. **Recursividad en Subárboles:**
    - Llamamos recursivamente a **best_ruta** para los subárboles izquierdo y derecho, actualizando los parámetros.

5. **Restauración de la Función:**
    - Eliminamos la ciudad actual de **ruta_actua**l para volver a antes, para procesar otros caminos.

Mohamed Amara El Houiti

Decrecimiento

En cada llamada recursiva, visitamos un nodo del árbol y luego los subárboles izquierdo y derecho, que cada vez son más pequeños que el árbol actual. Esto asegura que llegaremos a los nodos hoja y por lo tanto, tb llegaremos al final de la recursividad, ya q llegaremos a los casos base y la recursión se tb se acabará.

Código de la función

```
void Cuenca::bestRuta(const BinTree<string>& a, int comprar, int idcomprar, int vender, int idvender, vector<string>& rutaActual, int acumulado, int longfinal, ve
    if (!a.empty()) {
        int totalCompradas = 0;
        int totalVendidas = 0;

        if (inventarios[a.value()].productoExiste(idcomprar)) {
            int sobrante = inventarios[a.value()].cantidadTiene(idcomprar) - inventarios[a.value()].cantidadNecesita(idcomprar);
            if (sobrante > 0) {
                int cantidadComprar = min(comprar, sobrante);
                comprar -= cantidadComprar;
                totalCompradas += cantidadComprar;
            }
        }

        if (inventarios[a.value()].productoExiste(idvender)) {
            int sobrante = inventarios[a.value()].cantidadNecesita(idvender) - inventarios[a.value()].cantidadTiene(idvender);
            if (sobrante > 0) {
                int cantidadVender = min(vender, sobrante);
                vender -= cantidadVender;
                totalVendidas += cantidadVender;
            }
        }

        int cantidad = totalCompradas + totalVendidas;
        rutaActual.push_back(a.value());

        if (acumulado + cantidad > bestcantidadprods) {
            mejorRuta = rutaActual;
            bestcantidadprods = acumulado + cantidad;
            mejorLongitud = longfinal;
        } else if (acumulado + cantidad == bestcantidadprods) {
            if (longfinal < mejorLongitud) {
                mejorRuta = rutaActual;
                mejorLongitud = longfinal;
                bestcantidadprods = acumulado + cantidad;
            }
        }

        bestRuta(a.left(), comprar, idcomprar, vender, idvender, rutaActual, acumulado + cantidad, longfinal + 1, mejorRuta, bestcantidadprods, mejorLongitud);
        bestRuta(a.right(), comprar, idcomprar, vender, idvender, rutaActual, acumulado + cantidad, longfinal + 1, mejorRuta, bestcantidadprods, mejorLongitud);

        rutaActual.pop_back();
    }
}
```