Justificación de la práctica

Joel Márquez Álvarez

En este documento, se presentará la justificación de dos operaciones fundamentales en la implementación de la práctica de PRO2 de la Primavera de 2024. Las operaciones en cuestión son parte integral de la gestión de inventarios y rutas comerciales en una simulación de cuenca fluvial, con ciudades interconectadas que buscan intercambiar productos. A continuación, se detallarán las especificaciones, el código, y las justificaciones formales de las siguientes operaciones:

Operación Iterativa: Comerciar entre dos ciudades.
En esta operación, se actualizarán los inventarios de dos ciudades. Esta operación iterativa requiere un invariante de bucle bien definido y una justificación paso a paso, que garantice tanto la corrección como la finalización del proceso iterativo.

Operación Recursiva: Cálculo de la ruta en la operación hacer viaje
Dentro de la operación hacer_viaje, se utiliza una operación auxiliar recursiva para calcular la ruta óptima. Esta justificación incluye las hipótesis de inducción de las llamadas recursivas y una explicación detallada siguiendo los pasos formales para asegurar la corrección y finalización de la función, considerando el caso base y recursivo.

Para ambas operaciones, se ha seguido el enfoque del documento "*Correctesa de Programes Iteratius i Programació Recursiva"*, asegurando que todas las justificaciones incluyan invariantes de bucle (para la parte iterativa) y funciones de cota o equivalentes (para asegurar la finalización de ambas funciones).

El objetivo de este documento es proporcionar una demostración clara y detallada de la corrección de las implementaciones dadas, facilitando la comprensión y verificación de las operaciones realizadas dentro del código.

A continuación, se detallarán las especificaciones y el código de las operaciones a justificar, seguido por la demostración final detallada paso a paso.

1. Operación de comerciar

En esta sección, se presenta la justificación de la operación iterativa comerciar, que actualiza los inventarios de dos ciudades. Se encarga de equilibrar los inventarios de dos ciudades intercambiando productos excedentes y deficitarios.

Para ello, se recorre simultáneamente los inventarios de ambas ciudades, identificando productos coincidentes y ajustando las cantidades disponibles en función de las necesidades y excedentes respectivos. A continuación, se presenta la especificación y el código de la operación comerciar.

```
// Pre: El parámetro implícito y la ciudad c2 están correctamente inicializados y sus
inventarios contienen productos con IDs válidos y consistentes respecto al conjunto de
productos, que debe contener información válida sobre los productos, de sus pesos y volúmenes.

// Post: Se han intercambiado los productos que le sobran a una ciudad y que necesite la otra.
Los inventarios de ambas ciudades se han actualizado. Los atributos de peso y volumen total de
ambas ciudades se han ajustado adecuadamente.
```

```
void Ciudad::comerciar(Ciudad& c2, const Cjt_productos& cp) {
    auto it1 = _inv.begin(); // Iterador para la primera ciudad.
    auto it2 = c2._inv.begin(); // Iterador para la segunda ciudad.

    // Recorremos ambos inventarios simultáneamente.
    while (it1 != _inv.end() and it2 != c2._inv.end()) {
        // Si los productos coinciden en ambos inventarios:
        if (it1->first == it2->first) {
            int excedente1 = it1->second._prod_tiene - it1->second._prod_necesita;
            int excedente2 = it2->second._prod_tiene - it2->second._prod_necesita;

            // Si a la primera ciudad le sobran y a la segunda le faltan:
            if (excedente1 > 0 and excedente2 < 0) {
                // Actualizamos los inventarios y atributos de ambas ciudades
                int min_balance = min(excedente1, abs(excedente2));
                it1->second._prod_tiene -= min_balance;
                it2->second._prod_tiene += min_balance;
                int volumen = cp.consultar_volumen_producto(it1->first);
                int peso = cp.consultar_peso_producto(it1->first);

                _peso_total -= min_balance * peso;
                _volumen_total -= min_balance * volumen;
                c2._peso_total += min_balance * peso;
                c2._volumen_total += min_balance * volumen;
            }
            // Si a la primera ciudad le faltan y a la segunda le sobran:
            else if (excedente1 < 0 and excedente2 > 0) {
                // Actualizamos los inventarios y atributos de ambas ciudades
                int min_balance = min(abs(excedente1), excedente2);
                it2->second._prod_tiene -= min_balance;
                it1->second._prod_tiene += min_balance;
                int volumen = cp.consultar_volumen_producto(it1->first);
                int peso = cp.consultar_peso_producto(it1->first);

                c2._peso_total -= min_balance * peso;
```

```
                c2._volumen_total -= min_balance * volumen;
                _peso_total += min_balance * peso;
                _volumen_total += min_balance * volumen;
            }
            ++it1; // Avanzamos ambos iteradores.
            ++it2;
        }
        // Si la ID del producto de la primera ciudad es menor que el de la segunda, avanzamos
el iterador de la primera.
        else if (it1->first < it2->first) {
            ++it1;
        }
        // Si la ID del producto de la segunda ciudad es menor que el de la segunda, avanzamos
el iterador de la segunda.
        else {
            ++it2;
        }
```

1.1. Definición del invariante

Empezaremos definiendo el invariante. Matemáticamente, el invariante se puede expresar de la siguiente manera:

Para todo i tal que 0 <= i < *distancia(begin(inv), it1) y para todo j tal que 0 <= j < distancia(begin(c2.inv),it2)*, se cumple que:

comerciado(_inv[i], c2_inv[j])

- *begin(_inv)* es el iterador al inicio del inventario del parámetro implícito.
- *begin(c2._inv)* es el iterador al inicio del inventario de la ciudad c2.
- *distancia(a,b)* denota la distancia entre los iteradores a y b.
- *comerciado(_inv[i],c2._inv[j])* indica que los productos en las posiciones i y j de los inventarios han sido correctamente procesados y, si era necesario, se ha realizado el comercio entre ambas ciudades para esos productos.

En lenguaje natural, se deben cumplir estos tres puntos:

1) it1 apunta a un elemento del inventario del parámetro implícito (_inv) o a _inv.end().
2) it2 apunta a un elemento del inventario de la ciudad c2 (c2._inv) o a c2._inv.end().
3) Todos los elementos anteriores a it1 en el inventario del parámetro implícito y todos los elementos anteriores a it2 en el inventario de la ciudad c2 han sido tratados, y se ha realizado el comercio entre las dos ciudades para esos elementos, si era necesario.

Podemos ver que el invariante se cumple al principio con la precondición definida.

1.2. Inicializaciones

La razón para inicializar it1 con _inv.begin() e it2 con c2._inv.begin() es garantizar que, desde el principio del bucle, ambos iteradores apunten al primer elemento de los inventarios de sus respectivas ciudades. Esto permite procesar los inventarios en un orden secuencial y asegurarse de que no se omite ningún elemento, y que siempre se ha comerciado con los productos anteriores. Por ende, es necesario inicializarlo así para cumplir el invariante “todos los elementos anteriores han sido procesados”.

1.3. Condición de salida

```
// while (it1 != _inv.end() and it2 != c2._inv.end()) {
   // Cuerpo del bucle
}
```

El bucle continúa iterando mientras it1 no haya alcanzado el final del inventario del parámetro implícito (_inv.end()) y it2 no haya alcanzado el final del inventario de la ciudad c2 (c2._inv.end()). Esta condición garantiza que el bucle procesará todos los elementos de ambos inventarios hasta que uno de los iteradores haya recorrido completamente su respectivo inventario.

Debido a que ambos inventarios están representados por mapas ordenados, los iteradores it1 e it2 avanzan en orden ascendente de las claves (IDs de los productos). Esta propiedad de los mapas asegura que los elementos correspondientes en ambos inventarios se comparan y procesan de manera coordinada. Al iterar secuencialmente y procesar cada par de elementos, el bucle garantiza que todos los elementos en ambos inventarios serán considerados y comerciados según sea necesario.

Para justificar que la condición de salida garantiza el cumplimiento del invariante y la postcondición, consideremos los dos posibles casos de salida del bucle (necesariamente no puede haber más casos por cómo definimos nuestro bucle):

Caso 1: `it1 llega a _inv.end()`

Esto implica que it1 ha recorrido todos los elementos del inventario del parámetro implícito.

Por ende, han sido procesados y comerciados cuando ha sido necesario: no quedan más elementos en el inventario _inv para procesar, de manera que el invariante se cumple al final del bucle..

Es decir, la condición it1 == _inv.end() asegura que se han considerado todos los productos del inventario del parámetro implícito.

Caso 2: `it2 llega a _inv.end()`

Por simetría con el caso anterior, intercambiando el parámetro implícito por c2, podemos observar que se mantiene el invariante.

Vistos estos dos casos, que son los únicos posibles, veremos la parte de dentro: el cuerpo del bucle.

1.4. Cuerpo del bucle

Dentro del bucle *while*, el código maneja la lógica de comercio entre las dos ciudades, todo esto mientras it1 no haya alcanzado el final del inventario del parámetro implícito (_inv.end()) y it2 no haya alcanzado el final del inventario de la ciudad c2 (c2._inv.end()), como se ha explicado antes.

Primera ramificación: `if (it1->first == it2->first)`

En este bucle, la primera condición verifica si los productos actuales señalados por it1 e it2 son los mismos, verificando sus ID's en el mapa. Si los productos son iguales, se procede a calcular los excedentes de cada producto en ambas ciudades. El excedente se calcula restando la cantidad de producto necesario de la cantidad de producto disponible, información del *value* del mapa que es un *struct*.

Para hacer cálculos y comparaciones, utilizaremos la función *min* para encontrar el mínimo, que será lo que podremos comerciar, y la función *abs*, que nos dejará hacer la comparación del valor absoluto de ambos valores de manera que sea correcta de acorde al contexto.

Caso 1: `if (excedente1 > 0 and excedente2 < 0)`

Si la primera ciudad tiene un excedente positivo (excedente1 > 0) y la segunda ciudad tiene un déficit (excedente2 < 0), se determina el mínimo balance (min_balance) que se puede transferir entre las ciudades. La transferencia se realiza para ajustar las cantidades de producto en ambas ciudades, asegurando que una ciudad solo puede vender hasta lo que le sobra y comprar hasta el límite de lo que necesita. Se actualizan los inventarios y los atributos de peso y volumen de ambas ciudades usando los valores obtenidos del conjunto de productos (cp), llamando a sus métodos públicos que darán información correcta (basándonos en nuestra precondición).

Caso 2: `else if (excedente1 < 0 and excedente2 > 0)`

En el caso contrario, si la primera ciudad tiene un déficit (excedente1 < 0) y la segunda ciudad tiene un excedente (excedente2 > 0), se realiza una operación similar. Se transfiere la cantidad mínima posible del excedente de la segunda ciudad al déficit de la primera ciudad, actualizando nuevamente los inventarios y los atributos de peso y volumen de ambas ciudades. Los dos casos restantes, excedente1, excedente2 > 0 y excedente1, excedente2 < 0, no se consideran a la hora de comerciar ya que o bien ambas tienen lo que necesitan o ningúna, de manera que no pueden comerciar.

En ambos casos, después de procesar el comercio, se avanza ambos iteradores ++it1 y ++it2. Esto asegura que el siguiente par de elementos no procesados será considerado en la próxima iteración, manteniendo el invariante.

Segunda y tercera ramificación: `else if (it1->first < it2->first)` y `else`

La condición implícita de nuestro else es `it1->first > it2->first,` obtenida de negar los anteriores *ifs*. Si los productos señalados por it1 e it2 no son iguales, el código avanza el iterador correspondiente al producto menor, asegurando así que ambos inventarios se recorren de manera eficiente y ordenada, ya que los inventarios están ordenados y se van compensando a medida que se avanza. Esto asegura que it1 apunte al siguiente elemento en _inv que podría ser igual o mayor que el producto en it2, manteniendo el orden y el invariante (lo mismo sucede para it2).

En resumen, hemos visto que el cuerpo de nuestro bucle cumple el invariante en cada paso y se adecúa a las necesidades descritas.

1.5. Terminación

La terminación del bucle *while* en la función comerciar está garantizada porque, en cada iteración del bucle, al menos uno de los iteradores (it1 o it2) se avanza. Esta operación asegura que nos acercamos progresivamente al final de los inventarios de ambas ciudades. A continuación, se justifica formalmente la terminación del bucle utilizando una función de cota.

$$C(it1,it2)=(\_inv.end() - it1)+(c2.\_inv.end() - it2)$$

- *_inv.end()* es el iterador que apunta al final del inventario del parámetro implícito.
- *c2._inv.end()* es el iterador que apunta al final del inventario de la ciudad c2.
- *it1* es el iterador actual del inventario del parámetro implícito.
- *it2* es el iterador actual del inventario de la ciudad c2.

Para garantizar que el bucle termina, debemos demostrar que la función de cota es monotónicamente decreciente. Esto significa que en cada iteración del bucle, su valor debe reducirse. Distinguimos tres posibilidades:

Caso 1: `if (it1->first == it2->first), entonces se ejecuta ++it1, ++it2.`

$$C(it1+1, it2+1) = (\_inv.end() - (it1+1)) + (c2.\_inv.end() - (it2+1)) = C(it1, it2) - 2$$

Esto reduce la función de cota en 2, ya que ambos iteradores avanzan. Aclaración: por la condición del bucle, garantizamos que no nos excedemos del .end() nunca.

Caso 2: `if (it1->first < it2->first), entonces se ejecuta ++it1`

$$C(it1+1, it2) = (\_inv.end() - (it1+1)) + (c2.\_inv.end() - it2) = C(it1, it2) - 1$$

Se reduce la función de cota en 1, ya que it1 avanza.

Caso 3: `if (it1->first > it2->first), entonces se ejecuta ++it2`

Por simetría con el caso anterior, vemos que también la función de cota se reduce en 1.

En todos los casos, la función de cota disminuye con cada iteración del bucle. Dado que la función de cota es una suma de distancias que son finitas y no negativas, y dado que es decreciente y termina siendo cero cuando ambos iteradores alcanzan el final de sus respectivos inventarios, podemos concluir que el bucle necesariamente terminará.

Por lo tanto, el bucle es finito y se garantiza su terminación, cumpliendo así con los requisitos del algoritmo y asegurando que todas las condiciones de comercio se aplican a todos los productos de ambos inventarios, cumpliendo la postcondición.

2. Operación para calcular la ruta

En esta sección, se presenta la justificación de la operación recursiva encontrar_camino, función auxiliar de hacer_viaje que optimiza la ruta de un barco para maximizar el comercio entre barco y ciudades siguiendo una estructura en forma de árbol binario. La función equilibra las transacciones comerciales del barco, permitiendo la compra y venta de productos en cada ciudad visitada.

Se recorre el árbol de manera recursiva, evaluando en cada nodo (ciudad) si se pueden comprar o vender productos según la necesidad y el excedente de cada ciudad. La función elige la mejor ruta que maximiza la cantidad total de productos comprados y vendidos. En caso de empate, se selecciona el camino más corto. Se actualiza la lista de ciudades visitadas y las unidades comerciadas en cada paso, devolviendo finalmente el camino óptimo y el total de unidades compradas y vendidas. A continuación, se presenta la especificación y el código de la operación encontrar_camino:

```
// Pre: Hacemos las siguientes suposiciones:
1) `t` es un árbol binario de ciudades, que puede estar vacío.
2) `b` es un barco que contiene información sobre la cantidad de productos que puede comprar y
vender.
3) `compradas` es el número de unidades de productos comprados hasta el momento. Inicialmente
es 0.
4) `vendidas` es el número de unidades de productos vendidos hasta el momento. Inicialmente 0.
5) `ruta` es una lista de elementos de camino (ciudad, unidades compradas y vendidas).
Inicialmente está vacía.

// Post: La función auxiliar se encarga de cumplir lo siguiente que requiere hacer_viaje:
1) Se ha encontrado el camino que cumple las condiciones pedidas:
que maximiza las unidades de productos comprados y vendidos, y en caso de empate, el camino
más corto (y por último, el más a la izquierda).
2) `ruta` contiene la secuencia de ciudades por las que pasa el barco, junto con las unidades
de productos comprados y vendidos en cada ciudad.
3) Se devuelve un par donde el primer valor es el total de unidades compradas y el segundo
valor es el total de unidades vendidas en el camino encontrado.

pair<int,int> Cuenca::encontrar_camino(const BinTree<string>& t, Barco& b, int compradas, int
vendidas, list<ElementoCamino>& ruta) {
    if (t.empty() or (compradas == b.consultar_num_comprar() and vendidas ==
b.consultar_num_vender())) {
        ruta = list<ElementoCamino>();
        return make_pair(0,0);
    } else {
        auto it = _lista_ciudades.find(t.value());
        int unidades_c = 0;
        int unidades_v = 0;

        if ((*it).second.hay_prod_ciudad(b.consultar_id_prod_comprar())) {
            int sobra_ciudad_c =
(*it).second.consultar_sobra_ciudad(b.consultar_id_prod_comprar());
            if(sobra_ciudad_c > 0){ // Le sobran, podemos comprar.
                if (sobra_ciudad_c <= b.consultar_num_comprar()-compradas) {
                    unidades_c = sobra_ciudad_c;
                } else {
                    unidades_c = b.consultar_num_comprar()-compradas;
                }
            }
        }
```

```
        if ((*it).second.hay_prod_ciudad(b.consultar_id_prod_vender())) {
            int sobra_ciudad_v =
(*it).second.consultar_sobra_ciudad(b.consultar_id_prod_vender());
            if(sobra_ciudad_v < 0){ // Le faltan, podemos vender.
                sobra_ciudad_v = -sobra_ciudad_v;
                if (sobra_ciudad_v <= b.consultar_num_vender()-vendidas) {
                    unidades_v = sobra_ciudad_v;
                } else {
                    unidades_v = b.consultar_num_vender()-vendidas;
                }
            }
        }

        list<ElementoCamino> ruta_izq, ruta_der;
        pair<int,int> res_izq = encontrar_camino(t.left(), b, compradas+unidades_c,
vendidas+unidades_v, ruta_izq);
        int sumaleft = res_izq.first+res_izq.second;
        pair<int,int> res_der = encontrar_camino(t.right(), b, compradas+unidades_c,
vendidas+unidades_v, ruta_der);
        int sumaright = res_der.first+res_der.second;

        pair<int,int> maximo_cv; // Máximo comprado y vendido

        // Escogemos la mejor ruta.
        if (sumaleft < sumaright) {
            maximo_cv = res_der;
            ruta = ruta_der;
        }
        else if(sumaleft > sumaright){
            maximo_cv = res_izq;
            ruta = ruta_izq;
        }
        else{
            if(ruta_izq.size() <= ruta_der.size()){
                maximo_cv = res_izq;
                ruta = ruta_izq;
            }
            else{
                maximo_cv = res_der;
                ruta = ruta_der;
            }
        }

        // Hacer push cuando solo cuando sea necesario.
        if (unidades_c > 0 or unidades_v > 0 or !ruta_esc.empty()) {
            ElementoCamino ec;
            ec.id_ciudad = t.value();
            ec.unidades_c = unidades_c;
            ec.unidades_v = unidades_v;
            ruta.push_front(ec);
        }

        // Actualizamos los productos que hemos comerciado.
        return make_pair(maximo_cv.first+unidades_c, maximo_cv.second+unidades_v);
    }
```

2.1. Definición de las hipótesis de inducción

Para definir las hipótesis de inducción de las llamadas recursivas a la izquierda y a la derecha en la función encontrar_camino, necesitamos aplicar el post de la función a las llamadas recursivas. Esto implica que tomamos las condiciones finales deseadas y las consideramos como verdaderas para los subárboles izquierdo y derecho. A continuación, se presentan las hipótesis de inducción para ambas llamadas recursivas:

- Hipótesis de Inducción (HI) para la llamada recursiva a la izquierda:

```
encontrar_camino(t.left(), b, compradas+unidades_c, vendidas+unidades_v, ruta_izq);
```

La función devuelve una lista, ruta_izq, que contiene la secuencia de ciudades que maximiza las unidades de productos comprados y vendidos hasta ese punto, y en caso de empate, la secuencia de ciudades más corta, a la izquierda si de nuevo empata. El par que declaramos para guardar la llamada, (res_izq.first, res_izq.second) representa el total de unidades compradas y vendidas en esa ruta.

- Hipótesis de Inducción (HI) para la llamada recursiva a la derecha:

```
encontrar_camino(t.right(), b, compradas+unidades_c, vendidas+unidades_v, ruta_der);
```

La función devuelve una lista ruta_der que contiene la secuencia de ciudades que maximiza las unidades de productos comprados y vendidos hasta ese punto, y en caso de empate, la secuencia de ciudades más corta. El par (res_der.first, res_der.second) representa el total de unidades compradas y vendidas en esa ruta.

Estas hipótesis de inducción se utilizan para determinar las mejores rutas posibles en los subárboles izquierdo y derecho. Comparando los resultados de res_izq y res_der junto con sus correspondientes rutas ruta_izq y ruta_der, se elige la mejor ruta posible según las condiciones especificadas La ruta seleccionada se actualiza con la información de la ciudad actual y se devuelve junto con el total de unidades compradas y vendidas.

Las hipótesis garantizan que las llamadas recursivas producen rutas óptimas en sus respectivos subárboles, permitiendo que la función completa también produzca una ruta óptima para el árbol completo.

2.2. Casos base

Se han fijado dos casos básicos diferentes: por un lado, miramos cuando el árbol está vacío, y por otro, nos fijamos que hemos vendido y comprado todo lo posible. Ahora desarrollaremos ambos casos de manera individual y justificando su existencia:

```
Caso 1: t.empty(), entonces ruta = list<ElementoCamino>() y return make_pair(0,0);
```

El primer caso ocurre cuando el árbol binario está vacío, lo que significa que no hay ciudades para visitar y, por lo tanto, no se pueden realizar transacciones comerciales. En este escenario, la ruta es una lista vacía y el total de productos comprados y vendidos es cero.

Esto se representa con el par (0, 0), que indica que ya se ha alcanzado el objetivo de compra y venta, y no se requiere comerciar más.

Caso 2: `compradas == b.consultar_num_comprar() and vendidas == b.consultar_num_vender()`

`entonces ruta = list<ElementoCamino>() y return make_pair(0,0);`

El segundo caso base se presenta cuando el barco ha alcanzado su objetivo de compraventa, es decir, ha comprado y vendido el número total de productos especificados. Esto lo sabemos usando las consultoras del barco y nuestras variables de compradas y vendidas.

La ruta, entonces, debe ser una lista vacía porque no se necesita continuar el viaje. El par (0, 0) indica que ya se ha alcanzado el objetivo de compra y venta, y no se requieren transacciones adicionales.

2.3. Caso recursivo

En el caso recursivo de la función encontrar_camino, se busca determinar y seleccionar la mejor ruta posible en los subárboles izquierdo y derecho del árbol binario t, comenzando desde la ciudad actual t.value(). A continuación, se describe y justifica el proceso de forma detallada, utilizando las hipótesis de inducción para asegurar que la ruta óptima se selecciona correctamente.

1) Cálculo inicial

Primero, se analiza la ciudad actual, t.value(), para determinar si se pueden realizar intercambios comerciales. Esto implica verificar si la ciudad dispone del producto que el barco desea comprar y si necesita el producto que el barco desea vender. Se accede a los datos de la ciudad en _lista_ciudades y se comparan las capacidades de compra y venta del barco b con las necesidades y excedentes de la ciudad.

- Compras: Si la ciudad tiene un excedente del producto que el barco desea comprar, se calcula cuántas unidades se pueden comprar, teniendo en cuenta tanto el excedente de la ciudad como la capacidad restante del barco para comprar productos.

- Ventas: De manera similar, si la ciudad necesita el producto que el barco desea vender, se calcula cuántas unidades se pueden vender, considerando tanto la necesidad de la ciudad como la capacidad restante del barco para vender productos.

Estos cálculos se almacenan en unidades_c (para compras) y unidades_v (para ventas). Toda la información la obtenemos de nuestras variables o bien de consultoras del barco o de la ciudad, que se asume que funcionan correctamente en la precondición.

2) Llamadas recursivas

Se realizan dos llamadas recursivas: una para el subárbol izquierdo y otra para el subárbol derecho:

```
pair<int,int> res_izq = encontrar_camino(t.left(), b, compradas+unidades_c,
                        vendidas+unidades_v, ruta_izq);
pair<int,int> res_der = encontrar_camino(t.right(), b, compradas+unidades_c,
                        vendidas+unidades_v, ruta_der);
```

Según nuestras hipótesis de inducción, asumimos que ruta_izq contiene la mejor ruta en el subárbol izquierdo y que res_izq contiene el total de unidades compradas y vendidas en esa ruta. De manera similar, ruta_der y res_der son correctas para el subárbol derecho. Esto nos permite confiar en que las llamadas recursivas devuelven rutas óptimas para los subárboles correspondientes.

3) Comparación de rutas

Una vez obtenidas las rutas y los resultados de las unidades compradas y vendidas de los subárboles izquierdo y derecho, se procede a compararlas para determinar cuál es la mejor:

```
int sumaleft = res_izq.first+res_izq.second;
int sumaright = res_der.first+res_der.second;
```

Se calcula la suma total de productos comprados y vendidos para las rutas izquierda (sumaleft) y derecha (sumaright).

Si la ruta izquierda tiene un total de productos comprados y vendidos mayor que la derecha, se selecciona la ruta izquierda. Si la derecha tiene un total mayor, se selecciona la derecha.

En caso de empate en el total de productos comprados y vendidos, se elige la ruta más corta. Además, si empatan en todo eso, la izquierda.

```
if (sumaleft < sumaright) {
            maximo_cv = res_der;
            ruta = ruta_der;
        }
        else if(sumaleft > sumaright){
            maximo_cv = res_izq;
            ruta = ruta_izq;
        }
        else{
            if(ruta_izq.size() <= ruta_der.size()){
                maximo_cv = res_izq;
                ruta = ruta_izq;
            }
            else{
                maximo_cv = res_der;
                ruta = ruta_der;
            }
        }
```

Esta comparación asegura que se selecciona la ruta que buscamos en la función hacer_viaje.

4) Actualización de la ruta

```
if (unidades_c > 0 or unidades_v > 0 or !ruta_esc.empty()) {
            ElementoCamino ec;
            ec.id_ciudad = t.value();
            ec.unidades_c = unidades_c;
            ec.unidades_v = unidades_v;
            ruta.push_front(ec);
        }
```

Esta parte del código verifica si se debe actualizar la ruta. Se asegura que la ciudad actual se añade a la ruta solo si se ha realizado una transacción (compra o venta) o si ya se ha iniciado una secuencia de visitas. Así evitamos añadir ciudades que no aporten nada, pero siendo conscientes de que si hay alguna intermedia con poca aportación en una buena ruta tenemos que añadirla (condición de !ruta_esc.empty())

Si cualquiera de estas condiciones se cumple, se crea un objeto llamado “ElementoCamino” que almacena la ciudad actual y las cantidades de productos comprados y vendidos. Este objeto se añade al inicio de la lista ruta mediante *push_fron*t. Esto es crucial, ya que estamos construyendo la ruta desde las hojas del árbol hacia la raíz recursivamente, y añadir cada ciudad al inicio de la lista mantiene el orden cronológico correcto del viaje.

De esta manera, la función garantiza que la ruta acumulada refleja con precisión las transacciones realizadas en cada ciudad visitada, contribuyendo a la construcción de la ruta ideal del barco.

5) Resultado final

```
return make_pair(ruta_mejor.first+unidades_c, ruta_mejor.second+unidades_v);
```

Finalmente, el resultado se actualiza y ya hemos modificado en ruta la nueva mejor ruta, junto con el total de unidades compradas y vendidas. Se devuelve un par que indica el total de productos comprados y vendidos en esta ruta.

La nueva ruta devuelta contiene la secuencia de ciudades pedida, y el par devuelto refleja el total de productos comprados y vendidos en dicha ruta. Vemos que efectivamente llegamos a la postcondición siguiendo este esquema recursivo.

2.4. Decrecimiento

El decrecimiento de la función encontrar_camino está garantizado por la estructura misma de su recursividad sobre el árbol binario t. Cada vez que la función se llama recursivamente, se hace sobre uno de los subárboles (izquierdo o derecho) de t, lo cual asegura que en cada llamada recursiva, la función trabaja con un árbol más pequeño que el árbol original.

Este proceso de llamar recursivamente a los hijos izquierdo y derecho asegura que cada invocación de la función trabaja con una parte más pequeña del árbol original. Esta reducción progresiva en el tamaño del árbol en cada llamada recursiva cumple con el criterio de decrecimiento necesario para que la recursión termine. Así, la estructura recursiva de la función garantiza que siempre se alcanzará el caso base de t.empty como mínimo (si no se alcanza el otro criterio antes).

Eventualmente, la función alcanzará un nodo hoja (un nodo sin hijos), y finalmente, el árbol vacío. Por lo tanto, hemos demostrado que decrece.

3. Conclusión

En conclusión, hemos explorado en detalle la justificación de una función iterativa de comercio entre ciudades y la función recursiva de encontrar el camino óptimo de un barco. Mediante una cuidadosa formulación paso a paso, hemos asegurado que el proceso de comercio se realiza de manera eficiente y precisa, comerciando correctamente las unidades compradas y vendidas.

La estructura recursiva de la función encontrar_camino, apoyada en sólidas hipótesis de inducción y justificaciones de decrecimiento, garantiza la selección del mejor camino posible, considerando tanto la cantidad de productos transaccionados como la longitud de la ruta.

Este enfoque de justificar ha sido útil para hacer modificaciones al código y mejorarlo, proporcionando una base robusta por si en algún momento fuera necesario cambiar algo.