Fet per: Pau Bru Ribes

Funció `comerciar`

Implementació

Per permetre el comerç entre dues ciutats, `comerciar` actualitza els inventaris de les ciutats implicades (una implícita i l'altra passada com a paràmetre) basant-se en les diferències disponibles o necessàries dels productes comuns. La funció itera a través dels inventaris de les ciutats, ajustant les quantitats de productes fins que ja no hi ha més comuns.

Especificació

```
/* Pre: La ciutat implícita i "other" són vàlides. El conjunt de productes no és buit. */
/* Post: Les dues ciutats han intercanviat productes si en tenien disponibles. */
void Ciutat::comerciar(Ciutat& other, const Cjt_productes& productes);
```

Codi

```
void Ciutat::comerciar(Ciutat& other, const Cjt_productes& productes) {
    // Les ciutats no tenen inventari (Optimització)
    // Les ciutats no tenen inventari (Optimització)
    if (not this->teInventari() or not other.teInventari()) return;

    // Iterators per recórrer els inventaris
    auto it1 = this->inventari.begin();taris
    // Iterators per recórrer els inventaris
    auto it1 = this->inventari.begin();
    auto it2 = other.inventari.begin();

    // Comerç entre les ciutats    while (it1!=this->inventari.end() and it2!=other.inventari.end()) {
        // Consultar els productes
        int prod_id1 = it1->first;
        int prod_id2 = it2->first;

        // Si els productes són iguals
        if (prod_id1 == prod_id2) {
            // Consultar les diferències de productes de cada ciutat
            int dif1 = it1->second.second;
            int dif2 = it2->second.second;

            // A la ciutat1 li falta, a la ciutat2 li sobra
            if (dif1<0 and dif2>0) {
                // Consultar el pes i volum del producte
                pair<int,int> pesVol = productes.consultarProducte(prod_id1);

                // Determinar la màxima quantitat a intercanviar
                int quantitat = min(abs(dif1), abs(dif2));

                // Actualitzar l'inventari de la ciutat1
                it1->second.first.first += quantitat;  // Afegir la nova quantitat d'oferta
                it1->second.second += quantitat;  // Actualitzar la necessitat del producte
                this->pes_total += quantitat*pesVol.first;  // Actualitzar el pes total
                this->volum_total += quantitat*pesVol.second;  // Actualitzar el volum total

                // Actualitzar l'inventari de la ciutat2
                it2->second.first.first -= quantitat;  // Treure la quantitat venuda
                it2->second.second -= quantitat;  // Actualitzar la necessitat del producte
                other.pes_total -= quantitat*pesVol.first;  // Actualitzar el pes total
                other.volum_total -= quantitat*pesVol.second;  // Actualitzar el volum total
            }
            // A la ciutat1 li sobra, a la ciutat2 li falta
            else if (dif1 > 0 and dif2 < 0) {
                // Consultar el pes i volum del producte
                pair<int,int> pesVol = productes.consultarProducte(prod_id1);

                // Determinar la màxima quantitat a intercanviar
                int quantitat = min(abs(dif1), abs(dif2));

                // Actualitzar l'inventari de la ciutat1
                it1->second.first.first -= quantitat;  // Treure la quantitat venuda
                it1->second.second -= quantitat;  // Actualitzar la necessitat del producte
                this->pes_total -= quantitat*pesVol.first;  // Actualitzar el pes total
                this->volum_total -= quantitat*pesVol.second;  // Actualitzar el volum total

                // Actualitzar l'inventari de la ciutat2
                it2->second.first.first += quantitat;  // Afegir la nova quantitat d'oferta
                it2->second.second += quantitat;  // Actualitzar la necessitat del producte
                other.pes_total += quantitat*pesVol.first;  // Actualitzar el pes total
                other.volum_total += quantitat*pesVol.second;  // Actualitzar el volum total
            }
            // Avançar als següents productes
            it1++;
            it2++;
        }
        else if (prod_id1<prod_id2) {
            // Avançar els productes de la ciutat1 fins arribar a un producte comú amb la ciutat2
            it1 = this->inventari.lower_bound(prod_id2);
        }
        else {
            // Avançar els productes de la ciutat2 fins arribar a un producte comú amb la ciutat1
            it2 = other.inventari.lower_bound(prod_id1);
        }
    }
}
```

Justificació

Invariant

Durant cada iteració del bucle, els iteradors `it1` i `it2` recorren els inventaris de `this` i `other`, respectivament, ajustant les quantitats de productes. Definim l'invariant com:
$I : \forall p \in Productes, (it1 < this\text{->}inventari.end()) \wedge (it2 < other\text{->}inventari.end())$

D'aquesta manera l'invariant garanteix que tots els productes presents en els inventaris de les ciutats seran considerats i si algun iterador equival a `inventari.end()` s'acaba l'execució del bucle.

1. Inicialitzacions

- Inicialment, es comprova si alguna de les dues ciutats no té inventari. Si aquest és el cas, es retorna immediatament donat que no es podrà realitzar cap intercanvi.
- Els iteradors `it1` i `it2` s'inicialitzen a `this->inventari.begin()` i `other->inventari.begin()`, respectivament. Això permet començar a comparar els productes des del principi dels inventaris.

2. Condició de sortida

El bucle `while` s'executa mentre els iteradors `it1` i `it2` no hagin arribat al final dels inventaris. La condició de sortida del bucle és `it1 == this->inventari.end()` o `it2 == other.inventari.end()`, la qual cosa significa que s'han processat tots els productes de, almenys, una de les dues ciutats.

3. Cos del bucle

- **Consultes de productes:** Es consulten els identificadors dels productes actuals (`prod_id1` i `prod_id2`).
- **Productes iguals (`prod_id1 == prod_id2`):** Si els productes són iguals, es consulten les diferències de quantitats de cada ciutat (`dif1` i `dif2`). Hi ha dos casos possibles:
  - **A la ciutat1 li falta i a la ciutat2 li sobra (`dif1<0 && dif2>0`):** Es determina la quantitat màxima a intercanviar, es consulta el pes i volum del producte, i es realitzen les actualitzacions pertinents en els inventaris i totals de pes i volum de les dues ciutats.
  - **A la ciutat1 li sobra i a la ciutat2 li falta (`dif1>0 && dif2<0`):** Es determina la quantitat màxima a intercanviar, es consulta el pes i volum del producte, i es realitzen les actualitzacions pertinents en els inventaris i totals de pes i volum de les dues ciutats.
  - **Avançar iteradors:** Després de realitzar les actualitzacions, els iteradors `it1` i `it2` s'incrementen per passar als següents productes.
- **Productes diferents (`prod_id1 < prod_id2`):** Si els productes són diferents, `it1` s'ajusta perquè apunti al producte més proper que sigui igual o superior a `prod_id2` utilitzant `lower_bound`.
- **Productes diferents (`prod_id1 > prod_id2`):** Si els productes són diferents, `it2` s'ajusta perquè apunti al producte més proper que sigui igual o superior a `prod_id1` utilitzant `lower_bound`.

4. Acabament

A cada iteració del bucle, almenys un dels iteradors (`it1` o `it2`) avança. Això assegura que la funció de fita decreix a cada iteració, garantint així que el bucle finalitza després d'un nombre finit de passos.

Funció de fita

La funció de fita és la quantitat d'elements restants a processar en els inventaris de les dues ciutats. Definim la funció de fita com:
Funció de fita: $(this\text{->}inventari.end() - it1) + (other\text{->}inventari.end() - it2)$

A cada iteració, almenys un dels iteradors (`it1` i/o `it2`) avança, de manera que el nombre total d'elements restants a processar disminueix.

Instruccions finals

Quan el bucle finalitza, els iteradors `it1` i `it2` han recorregut tots els elements dels inventaris de les dues ciutats. Les quantitats de mercaderies s'han ajustat segons les necessitats i excessos de cada ciutat, complint la post-condició establerta.

Funció `determinar_viatge`

Implementació

La funció `determinar_viatge` busca determinar el millor viatge possible maximitzant la quantitat de comerç entre el vaixell i les respectives ciutats. Això es fa explorant un arbre binari de ciutats (`cuenca`) i realitzant intercanvis amb un vaixell (`barco`). La funció utilitza la recursivitat per explorar totes les possibilitats i seleccionar la millor opció.

Especificació

```
/* Pre: cuenca és un arbre binari amb id's de ciutats vàlides, productes és un conjunt de productes vàlids i no buit, barco és un vaixell inicialitzat */
/* Post: Retorna el millor viatge possible maximitzant la quantitat de comerç realitzat */
Viatge Cjt_ciutats::determinar_viatge(const BinTree<string>& cuenca, const Cjt_productes& productes, Vaixell barco);
```

Codi

```
Viatge Cjt_ciutats::determinar_viatge(const BinTree<string>& cuenca, const Cjt_productes& productes, Vaixell barco) {
    // === Base Case
    if (cuenca.empty()) return Viatge();

    // === General Case
    Viatge viatge_act;
    string id_city = cuenca.value();

    // Fer intercanvi
    int quant_comerciat = barco.comerciarSenseMod(cmap[id_city]);

    // Actualitzar viatge actual
    viatge_act.afegirCiutat(id_city);
    viatge_act.actQuant(quant_comerciat);

    // Si el barco ja no té unitats per intercanviar, es para tot.
    if (barco.quantitatPerComprar()==0 and barco.quantitatPerVendre()==0) return viatge_act;

    // Recursivitat
    Viatge viatge_esquerra, viatge_dreta;
    // Explorar ciutat de l'esquerra
    if (not cuenca.left().empty()) viatge_esquerra = determinar_viatge(cuenca.left(), productes, barco);
    // Explorar ciutat de la dreta
    if (not cuenca.right().empty()) viatge_dreta = determinar_viatge(cuenca.right(), productes, barco);

    // Determinar millor viatge
    // Casos:
    // 1. No hi ha viatges a l'esquerra ni a la dreta
    if (viatge_esquerra.consultarQuant()==0 and viatge_dreta.consultarQuant()==0) return viatge_act;
    // 2. No hi ha comerç a l'esquerra
    else if (viatge_esquerra.consultarQuant()==0) {
        viatge_act.actViatge(viatge_dreta);
        return viatge_act;
    }
    // 3. No hi ha comerç a la dreta
    else if (viatge_dreta.consultarQuant()==0) {
        viatge_act.actViatge(viatge_esquerra);
        return viatge_act;
    }
    // 4. Hi ha comerç a ambdues bandes
    // 4.1. La quantitat de la esquerra és més gran
    else if (viatge_esquerra.consultarQuant()>viatge_dreta.consultarQuant()) {
        viatge_act.actViatge(viatge_esquerra);
        return viatge_act;
    }
    // 4.1 Tenen la mateixa quantitat
    else if (viatge_esquerra.consultarQuant()==viatge_dreta.consultarQuant()) {
        // 4.1.1 La distància de la esquerra és més petita o igual
        if (viatge_esquerra.consultarDist()<=viatge_dreta.consultarDist()) {
            viatge_act.actViatge(viatge_esquerra);
            return viatge_act;
        }
    }
    // 4.2. La quantitat de la dreta és més gran
    viatge_act.actViatge(viatge_dreta);

    return viatge_act;
}
```

Justificació

Invariant

Durant cada crida recursiva de la funció, es manté l'invariant que les ciutats explorades fins al moment estan correctament processades, i les decisions de comerç són òptimes respecte a les condicions donades. Definim l'invariant com:

Invariant: $I : \forall node \in cuenca, viatge\_act$ *representa el millor viatge possible fins a aquest punt*

1. Inicialitzacions

- Si `cuenca` està buit (`cuenca.empty()`), es retorna un viatge buit (`Viatge()`). Això cobreix el cas base de la recursió.
- Es crea un objecte `Viatge` buit anomenat `viatge_act` per representar el viatge actual.
- S'agafa el valor del node actual de `cuenca` (`id_city`).

2. Intercanvi i actualització del viatge

- Es realitza un intercanvi de mercaderies amb la ciutat actual (`cmap[id_city]`) utilitzant el vaixell (`barco.comerciarSenseMod`), i es guarda la quantitat comerciada. Aquesta funció no requereix de la classe `Cjt_productes` donat que no modificarà la ciutat (Millorar eficiència).
- S'afegeix la ciutat al viatge actual (`viatge_act.afegirCiutat(id_city)`) i s'actualitza la quantitat comerciada (`viatge_act.actQuant(quant_comerciat)`). Aquesta pot ser 0, però més endavant discutirem perquè es continua ficant a la llista.
- Si el vaixell ja no té unitats per intercanviar (`barco.quantitatPerComprar() == 0 and barco.quantitatPerVendre() == 0`), es retorna el viatge actual.

3. Exploració recursiva

- Es defineixen dos objectes `Viatge` per als viatges de l'esquerra (`viatge_esquerra`) i la dreta (`viatge_dreta`).
- Si hi ha una ciutat a l'esquerra (`not cuenca.left().empty()`), es crida recursivament la funció `determinar_viatge` per explorar aquesta ciutat.
- Si hi ha una ciutat a la dreta (`not cuenca.right().empty()`), es crida recursivament la funció `determinar_viatge` per explorar aquesta ciutat.

4. Determinació del millor viatge

- Es comparen els viatges obtinguts de l'esquerra i la dreta per determinar quin és el millor viatge basant-se en la quantitat de comerç realitzat i la distància recorreguda.
- Si no hi ha comerç ni a l'esquerra ni a la dreta (`viatge_esquerra.consultarQuant() == 0 and viatge_dreta.consultarQuant() == 0`), es retorna el viatge actual. Això farà que en la llista no hi hagi ciutats posteriors on son redundants donat que no hi ha comerç.
- Si només hi ha comerç a una banda (`viatge_esquerra.consultarQuant() == 0 o viatge_dreta.consultarQuant() == 0`), es selecciona aquesta banda.
- Si hi ha comerç a ambdues bandes, es comparen les quantitats comerciades i, si són iguals, les distàncies, per determinar el millor viatge. En ultima instancia, si tant la quantitat com la distància és la mateixa, es selecciona la que sigui més cap a la dreta mirant el riu cap amunt.

5. Acabament

A cada crida recursiva, l'arbre `cuenca` es redueix en mida, assegurant que la recursió finalitza després d'un nombre finit de passos.

Funció de fita

La funció de fita és la profunditat de l'arbre cuenca restant per explorar. Definim la funció de fita com: Funció de fita: *Funció de fita: profunditat(cuenca)*

Instruccions finals

Quan la recursivitat finalitza, el viatge resultant (`viatge_act`) ha estat actualitzat per reflectir el millor viatge possible basant-se en les quantitats comerciades i les distàncies recorregudes. Es compleix la postcondició de la funció, ja que el viatge retornat maximitza la quantitat de comerç realitzat després d'explorar totes les ciutats possibles dins de l'arbre `cuenca`.

Justificació Detallada

Definició dels símbols matemàtics utilitzats

- x: Paràmetre actual que inclou `cuenca`, `productes` i `barco`.
- $h(x, r)$: Funció que actualitza el viatge amb el resultat de la crida recursiva.
- r: Resultat de la crida recursiva, que és un viatge obtingut a partir d'un subarbre de `cuenca`.
- $d(x)$: Viatge buit.
- $Q(x)$: Precondició. Indica que `cuenca` és un arbre binari de ciutats vàlides, `productes` és un conjunt de productes vàlids i `barco` és un vaixell inicialitzat.
- $Q(g(x))$: Precondició per la crida recursiva. Indica que els subarbres esquerra o dreta de cuenca compleixen les precondicions necessàries per a la crida recursiva.
- $c(x)$: Condició del cas senzill. Indica que `cuenca.empty()` és cert.
- $\neg c(x)$: Condició del cas recursiu. Indica que `cuenca` no està buit.
- $R(x, d(x))$: Postcondició del cas senzill. Indica que es retorna un viatge buit.
- $R(x, h(x, r))$: Postcondició del cas recursiu després d'actualitzar el viatge. Indica que el viatge actualitzat compleix la postcondició.
- $g(x)$: Funció que transforma l'entrada actual en l'entrada de la crida recursiva. Indica els subarbres esquerra o dreta de `cuenca`.
- $R(g(x), r)$: Postcondició per la crida recursiva. Indica que el millor viatge a l'esquerra o dreta compleix la postcondició.
- $t(x)$: Funció de fita. Indica la profunditat de l'arbre `cuenca`.
- $t(g(x)) < t(x)$: Condició de decreixement. Indica que la profunditat de l'arbre disminueix en cada crida recursiva.

Cas Sencill

Si `cuenca` està buit, la funció retorna un viatge buit (`Viatge()`), el qual compleix la postcondició ja que no hi ha ciutats per visitar ni comerç a realitzar.

Raonament: Si cuenca és buit, no hi ha ciutats per visitar ni comerç a realitzar. Això compleix la postcondició ja que el viatge resultant és correcte.

Matemàticament: $Q(x) \wedge c(x) \Rightarrow R(x, d(x))$

Cas Recursiu

1. Inicialitzacions

- Es crea un objecte Viatge buit anomenat viatge_act per representar el viatge actual.
- S'agafa el valor del node actual de cuenca (id_city).
- Es realitza un intercanvi de productes amb la ciutat actual (`cmap[id_city]`) utilitzant el vaixell (`barco.comerciarSenseMod`).
- S'afegeix la ciutat al viatge actual (`viatge_act.afegirCiutat(id_city)`) i s'actualitza la quantitat comerciada (`viatge_act.actQuant(quant_comerciat)`).

2. Intercanvi i actualització del viatge

Si el vaixell ja no té unitats per intercanviar (`barco.quantitatPerComprar() == 0 and barco.quantitatPerVendre() == 0`), es retorna el viatge actualitzat (viatge_act).

Raonament: Això significa que no hi ha més comerç possible, el viatge actual és el millor possible fins ara.

Matemàticament: $Q(x) \wedge \neg c(x) \Rightarrow R(x, h(x, r))$

3. Exploració recursiva

- Es defineixen dos objectes `Viatge` per als viatges de l'esquerra (`viatge_esquerra`) i la dreta (`viatge_dreta`).
- Si hi ha una ciutat a l'esquerra (`not cuenca.left().empty()`), es crida recursivament la funció determinar_viatge per explorar aquesta ciutat.
- Si hi ha una ciutat a la dreta (`not cuenca.right().empty()`), es crida recursivament la funció determinar_viatge per explorar aquesta ciutat.

Raonament: Això permet explorar totes les possibilitats de comerç a l'arbre `cuenca`.

Matemàticament: $Q(x) \wedge \neg c(x) \Rightarrow Q(g(x))$

4. Determinació del millor viatge

- **Cas 1:** Si no hi ha viatges comercials a l'esquerra ni a la dreta (`viatge_esquerra.consultarQuant() == 0 and viatge_dreta.consultarQuant() == 0`), es retorna viatge_act.
- **Cas 2 i Cas 3:** Si només hi ha comerç a una banda (`viatge_esquerra.consultarQuant() == 0 o viatge_dreta.consultarQuant() == 0`), es selecciona aquesta banda.
- **Cas 4:** Hi ha comerç a ambdues bandes
  - **Cas 4.1:** Si la quantitat de comerç a l'esquerra és major (`viatge_esquerra.consultarQuant() > viatge_dreta.consultarQuant()`), es selecciona `viatge_esquerra`.
    - **Cas 4.1.1:** Si la quantitat comerciat es la mateixa (`viatge_esquerra.consultarQuant()==viatge_dreta.consultarQuant()`) es selecciona la de menor distancia. En cas d'empat en distancia (`viatge_esquerra.consultarDist() <=viatge_dreta.consultarDist()`) es selecciona l'esquerra.
  - **Cas 4.2:** Altrament s'agafa el viatge de la subruta dreta.

Raonament: Això assegura que el viatge seleccionat maximitza la quantitat de comerç i minimitza la distància recorreguda.

Matemàticament: $Q(g(x)) \wedge R(g(x), r) \Rightarrow R(x, h(x, r))$

5. Decreixement

- A cada crida recursiva, l'arbre `cuenca` es redueix en mida, assegurant que la recursió finalitza després d'un nombre finit de passos.

Matemàticament: $Q(x) \Rightarrow t(x) \in \mathbb{N}\, Q(x) \wedge \neg c(x) \Rightarrow t(g(x)) < t(x)$

Instruccions Finals

Quan la recursivitat finalitza, el viatge resultant (`viatge_act`) ha estat actualitzat per reflectir el millor viatge possible basant-se en les quantitats comerciades i les distàncies recorregudes. Es compleix la postcondició de la funció, ja que el viatge retornat maximitza la quantitat de comerç realitzat després d'explorar totes les ciutats possibles dins de l'arbre `cuenca`.