JUSTIFICACIÓ D'ALGORITMES DE LA PRÀCTICA DE PRO2

1. Algoritme iteratiu - `City::TradeWith(City& other)` a `City.cc`

Nota: El codi es pot trobar després de la justificació.

Precondició: Les dues ciutats (*this* i *other*) són ciutats vàlides.
Postcondició: Les dues ciutats han fet tots els intercanvis possibles de productes.

Aclariment: ***C****ada ciutat té un iterador que apunta a les posicions d'un* `set` *que conté els identificadors naturals, sense repeticions, dels productes disponibles a la ciutat. Les ciutats només poden comerciar productes amb el mateix identificador.*

Invariant: els dos iteradors `this_product_it` i `other_product_it` apunten a posicions vàlides de memòria. Tots els intercanvis possibles de productes amb identificadors més petits que `this_product_it` i `other_product_it` han estat fets. En la i-èsima iteració, el valor de `this_product_it` és major que qualssevol valor que hagi estat iterat anteriorment per `other_product_it`. El recíproc també es compleix.

Aclariment: *Com que els elements d'un set no tenen cap índex dins del conjunt, donar la funció de fita com a una equació és complicat. No obstant, internament un set ordena els seus elements, pel que utilitzarem el mètode* ***fictici*** `set::getpos(iterator)` *per a referir-nos a la posició interna d'un element al set, començant pel zero i acabant a la posició mida-1.*

Funció de fita:

```
this_product_set.size() - this_product_set.getpos(this_product_it) – 1 +
other_product_set.size() - other_product_set.getpos(other_product_it) - 1
```

Justificació de l'algoritme:

Assumint que la precondició es compleix, les dues ciutats contenen un inventari vàlid, i, per tant, cada ciutat té un iterador que apunta a les posicions d'un `set` que conté els identificadors enters dels productes disponibles a la ciutat. Degut al funcionament del set de la STL, els iteradors iteraran a través del valors del set de menor a major, sense valors repetits.

En cas que, ja des del principi, no es compleixi la condició del bucle i algun dels iteradors apunti a una posició no vàlida de la memòria (és a dir, a la posició `end()` del set), voldrà dir que alguna de les ciutats no té cap producte disponible, i, en conseqüència, tots els intercanvis possibles (que serien zero) ja haurien estat fets, i, per tant, la postcondició es compliria. En aquest cas no caldria executar cap iteració del bucle, i es finalitzaria l'execució directament.

Altrament, en cas que l'invariant es compleixi, i mentre la condició del bucle es compleixi, els dos iteradors apuntaran a dos identificadors de producte vàlids i existents. Llavors, es poden donar 3 casos en una iteració qualsevol.

Cas 1) El primer identificador és major que el segon identificador: Això vol dir que la primera ciutat no conté el producte de `other_product_it`.

Ho veiem: Per l'invariant, podem deduir que la primera ciutat no conté el producte al que apunta `other_product_it`. Com que `other_product_it` és major que qualsevol identificador ja iterat per `this_product_it` i alhora és menor que `this_product_it`, deduïm que no pot estar present a la llista de productes de la primera ciutat.

Per tant, es desplaça l'iterador de la segona ciutat a la següent posició. (`other_product_it++`). L'invariant es segueix complint, i la funció de fita s'ha decrementat en 1.

Cas 2) El primer identificador és menor que el segon identificador: S'aplica el mateix raonament que al **cas 1)**, però amb els papers canviats. Per tant, es desplaça l'iterador de la primera ciutat a la següent posició (`this_product_it++`). L'invariant es segueix complint, i la funció de fita s'ha decrementat en 1.

Cas 3) Els dos productes coincideixen: Això significa que les dues ciutats contenen el mateix producte, i, per tant, s'executa la mecànica de comerç entre les dues ciutats.

Finalment, es desplacen els dos iteradors a les seves respectives següents posicions, ja que el producte que s'acaba d'avaluar no s'ha de tornar a comprovar, ja que s'acaben de fer tots els intercanvis possibles amb ell. Com que es garanteix que no hi haurà elements repetits en un mateix inventari, cap dels dos iteradors tornarà a apuntar al mateix element, fent que l'invariant es torni a complir. A més a més, la funció de fita s'ha decrementat en 2.

Un cop es deixi de complir la condició del bucle, la postcondició es complirà automàticament, i podrem finalitzar la execució. Ho veiem:

En cas que un dels dos iteradors (per exemple, `this_product_it`) apunti a `end()`, per l'invariant podem deduir que el valor al qual apunta `other_product_it` és estrictament major que tots els elements iterats anteriorment per `this_product_it`, és a dir, per tots els identificadors presents a la primera ciutat. Per tant, serà impossible que pel valor de `other_product_it` ni que pels possibles valors futurs es pugui comerciar (ja que la primera ciutat no els contindrà), provant que tots els intercanvis possibles han estat ja fets. El mateix es pot aplicat amb el paper dels dos iteradors canviats.

En cas contrari, si els dos iteradors apunten a `end()`, la justificació és trivial: No hi ha més productes a processar en cap ciutat, i, per tant, ja s'han fet tots els intercanvis possibles.

Codi de la funció iterativa:

```
void City::TradeWith(City& other)
{
    auto& this_product_set = this->GetRawProductIds();
    auto& other_product_set = other.GetRawProductIds();

    auto this_product_it = this_product_set.begin();
    auto other_product_it = other_product_set.begin();

    while(this_product_it != this_product_set.end()
        && other_product_it != other_product_set.end())
    {
        if(*this_product_it > *other_product_it)
            other_product_it++;
        else if(*this_product_it < *other_product_it)
            this_product_it++;

        else //(*this_product_it == *other_product_it)
        {
            int product_id = *this_product_it;

            // this -> SELLER
            // other -> BUYER
            int trade_amount = min(
                this->GetProductExceedingAmount(product_id),
                other.GetProductMissingAmount(product_id)
            );
            if(trade_amount != 0)
            {
                this->WithdrawProductAmount(product_id, trade_amount);
                other.RestockProductAmount(product_id, trade_amount);
            }
            else
            {
                // other -> SELLER
                // this -> BUYER
                trade_amount = min(
                    other.GetProductExceedingAmount(product_id),
                    this->GetProductMissingAmount(product_id)
                );
                if(trade_amount != 0)
                {
                   other.WithdrawProductAmount(product_id, trade_amount);
                   this->RestockProductAmount(product_id, trade_amount);
                }
            }

            this_product_it++;
            other_product_it++;
        }
    }
}
```

2. Algoritme recursiu

Precondició: `current_location` és un node no nul, el valor del qual és una ciutat que existeix a la vall. `current_route` és un vector (buit o no) el contingut del qual es correspon amb la posició del node `current_location` a la vall.

Els valors `bought_amount`, `sold_amount` i `recently_skipped_cities` són enters positius o zero, i tenen respectivament la quantitat del producte anteriorment comprat, del producte anteriorment venut i de les ciutats que s'han saltat des de l'última ciutat on hi ha hagut algun tipus de comerç.

`test_ship` és un vaixell vàlid que conté els productes i les quantitats per a les quals es vol calcular la ruta més adequada, a més a més de portar un recompte de les unitats dels respectius productes disponibles després de recórrer la `current_route` fins a la posició actual. Per tant, el vaixell és coherent amb `bought_amount` i `sold_amount.`

`best_route` apunta a una estructura amb la informació d'una ruta arbitrària però amb valors vàlids.

Postcondició: S'ha analitzat totes les rutes possibles que surten del node actual i passen pels nodes del subarbre que parteix del node actual. Si alguna ruta que comença per `current_route` i acaba amb alguna combinació possible dels fills de l'arbre actual és millor que la ruta emmagatzemada a `best_route`, aquesta es substitueix per la mencionada millor ruta.

Funció de fita: La mida de l'arbre passat pel paràmetre `current_location`

Justificació:

Assumint que la precondició es compleix, podem afirmar que tots els paràmetres (excepte `best_route`) contenen una ruta (no cal saber quina) a mig executar, i són coherents entre ells.

Per tant, com que es garanteix que la posició (i per tant, la ciutat) del viatge és vàlida i existeix, primer es duu a terme una simulació de comerç entre el vaixell i la ciutat. Es calculen les quantitats de producte a comerciar, i s'actualitzen les variables `bought_amount` i `sold_amount`, així com també s'actualitzen les quantitats de producte disponibles al vaixell, però **no** es modifiquen les ciutats.

Finalment, si es detecta que no s'ha comerciat cap producte (això es duu a terme amb la variable auxiliar `traded_amount`), s'augmenta el recompte de ciutats que s'han saltat. Altrament, es reseteja el comptador.

Un cop *computada* la posició actual, es poden donar dos casos:

1. Cas Base: La posició actual és la última posició possible de la ruta actual. Això es pot comprovar o bé veient que els fills del node `current_location` estan buits o bé veient que el vaixell ja ha fet tot el comerç que podia fer. (no queden productes ni a comprar ni a vendre), la ruta es pot considerar completada.

Un cop una ruta està acabada, i després de fer les modificacions necessàries a la ruta per a simplificar-la si s'escau, es comprova si cal substituir `best_route` per la ruta actual, i, si cal, aquesta es substitueix. Per tant, es compleix la postcondició, i es pot finalitzar l'execució.

2. Cas recursiu: Si les condicions del cas base no es compleixen, podem assegurar que la ruta té continuitat tant per el fill esquerre del com pel fill dret de `current_location`. Llavors, cal continuar el càlcul de la ruta a través dels dos fills.

Casualment, després de computar la posició actual, els paràmetres tornen a complir la precondició (la única diferència és que els valors ara consideren que el vaixell ja ha navegat `current_location`). Per tant, per hipòtesi d'inducció, podem cridar a la pròpia funció per a continuar el càlcul de la ruta tant pel fill dret com pel fill esquerre de `current_location`. Com que els fills de `current_location` no poden tenir una mida major o igual a `current_location`, la funció de fita es veu decrementada.

Un cop finalitza la crida de les funcions recursives es compleix la postcondició de les crides recursives (per hipòtesi d'inducció), que és la mateixa postcondició que la del mètode actual. Per tant, es pot finalitzar la execució.

```
void Valley::TestRouteStep(vector<NavStep>& current_route,
        const BinTree<string>& current_location,
        int bought_amount,
        int sold_amount,
        int recently_skipped_cities,
        Ship& test_ship,
        Valley::RouteEvaluationResult &best_route)
{
    int buying_id = test_ship.BuyingProduct().GetId();
    int selling_id = test_ship.SellingProduct().GetId();

    int traded_amount = 0;
    auto& city = GetCity(current_location.value());

    // Calculate how much product is going to be bought from the city
    if(city.HasProduct(buying_id))
    {
      int amount_to_buy = min(
            city.GetProductExceedingAmount(buying_id),
            test_ship.BuyingProduct().GetMissingAmount()
       );
      test_ship.BuyingProduct().RestockAmount(amount_to_buy);
      traded_amount += amount_to_buy;
      bought_amount += amount_to_buy;
    }

    // Calculate how much product is going to be sold to the city
    if(city.HasProduct(selling_id))
    {
      int amount_to_sell = min(
            city.GetProductMissingAmount(selling_id),
            test_ship.SellingProduct().GetExceedingAmount()
       );
      test_ship.SellingProduct().WithdrawAmount(amount_to_sell);
      traded_amount += amount_to_sell;
      sold_amount += amount_to_sell;
    }
```

```
    // If the current city has tradable product, reset the skipped cities
counter
    // Otherwhise, increase it
    if(traded_amount > 0)
        recently_skipped_cities = 0;
    else
        recently_skipped_cities++;

    // If we have reached the end of the river
    // or if we have reached the limits of the ship's trading capacity
    if(current_location.right().empty()
        || current_location.left().empty()
        || (test_ship.BuyingProduct().GetMissingAmount() == 0
           && test_ship.SellingProduct().GetExceedingAmount() == 0))
    {
        while(recently_skipped_cities > 0 && !current_route.empty())
        {
            // If the first city has been skipped, since the first city does
not
            // have an asscoiated NavStep on current_route, attempting to
pop_back() would crash
            current_route.pop_back();
            recently_skipped_cities--;
        }

        Valley::RouteEvaluationResult result;
        result.route = current_route;
        // We need to add 1 to take into account the first city, which does
not
        // have associated any NavStep on current_route
        result.EffectiveLength = current_route.size() + 1 -
recently_skipped_cities;
        result.TotalTrades = bought_amount + sold_amount;

        if(result.TotalTrades > best_route.TotalTrades
            || (result.TotalTrades == best_route.TotalTrades
            && result.EffectiveLength < best_route.EffectiveLength))
        {
            best_route = result;
        }
    }
    else
    {
        // Continue testing through the left
        auto left_route = current_route;
        auto left_ship = test_ship.Copy();
        left_route.push_back(Valley::NavStep::Left);
        TestRouteStep(left_route, current_location.left(), bought_amount,
sold_amount, recently_skipped_cities, left_ship, best_route);

        // Continue testing through the right
        current_route.push_back(Valley::NavStep::Right);
        TestRouteStep(current_route, current_location.right(), bought_amount,
sold_amount, recently_skipped_cities, test_ship, best_route);
    }
}
```