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.

Invariant: els dos iteradors `this_product_it` i `other_product_it` apunten a posicions vàlides de memòria.

Funció de fita: 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. Dit això, la funció de fita seria

```
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:

Suposant que la precondició es compleix, les dues ciutats contenen un inventari vàlid, i, per tant, mitjançant el mètode `City::GetRawProductIds()`, podem obtenir un ***set<int>*** amb tots els productes disponibles de la ciutat. D'aquest set n'obtindrem l'iterador, que ens permetrà accedir de forma ordenada als identificadors dels productes disponibles de cada ciutat, de menor a major. Els dos iteradors, que apunten cadascun a l'inventari de la seva ciutat corresponent, inicialment apuntaran al primer element del set, és a dir, a l'identificador més petit (en valor numèric) d'entre tots els productes disponibles a la ciutat.

En cas que l'invariant no es compleixi, i algun dels iteradors apunti d'entrada 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 l'invariant es compleixi, els dos iteradors apuntaran a dos identificadors de producte vàlids i existents. Llavors, es poden donar 3 casos.

Cas 1) El primer identificador és menor que el segon identificador: Això vol dir que, com que els identificadors estan ordenats en ordre ascendent i els iteradors apunten inicialment al valor més petit, la segona ciutat no conté el primer producte. Si la segona ciutat contingués el primer producte, el seu identificador hauria hagut d'aparèixer abans de l'identificador de producte actual de la segona ciutat, cosa que no ha fet, pel que deduïm que la segona ciutat no pot contenir el producte al qual apunta el primer iterador.

Per tant, es desplaça l'iterador de la primera ciutat a la següent posició. (`this_product_it++`)

Cas 2) El primer identificador és major 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 segona ciutat a la següent posició (`other_product_it++`)

Cas 3) Els dos productes coincideixen: Això significa que les dues ciutats contenen el mateix producte, pel que és possible que el puguin comerciar. Per tant, s'avaluen els dos escenaris possibles: que la primera ciutat li vengui producte a la segona i vici-versa.

Nota: Es sobreentén que parlem del producte l'identificador del qual coincideix amb el del valor dels iteradors.

Per a comprovar si la primera ciutat li pot vendre producte a la segona ciutat, es calcula el mínim entre la quantitat de producte excedent de la primera ciutat i la quantitat de producte necessitat per la segona ciutat. Si aquest nombre és major que zero es procedeix a fer l'intercanvi, restant-li a la primera ciutat la quantitat prèviament calculada del producte, i sumant-li a la segona ciutat.

En cas que no s'hagi produït venda, també es comprova el cas contrari, és a dir, si la segona ciutat li pot vendre producte a la primera, i es repeteix el procediment mencionat al paràgraf anterior, però intercanviant el paper de 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. *Aquí acaba el tercer cas.*

En els tres casos, com a mínim un dels dos iteradors s'ha incrementat, augmentant la posició relativa dins del set a la que apunten. Per tant, la funció de fita s'ha reduït. Llavors, el bucle torna al principi i torna a comprovar la condició.

Mentre es compleixi l'invariant hi haurà possibles combinacions de productes per a comerciar, i quan es deixi de complir, que voldrà dir que una de les ciutats ja no té més productes disponibles al seu inventari per a comerciar, ja que ja s'hauran comprovat tots, i la postcondició es complirà, ja que si a una ciutat no té més productes no té cap altre intercanvi possible que els ja comprovats.

Codi de la funció:

```
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

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

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.

`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. La ruta (partint des del node base) en la que més productes s'intercanvien, i en cas d'empat la més curta, queda emmagatzemada al paràmetre passat per referència `best_route`.

Justificació:

Assumint que la precondició es compleix, podem afirmar que tots els paràmetres (excepte `best_route`) contenen una ruta 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, primer cal eliminar totes les ciutats del final amb les que no s'ha comerciat. Això es fa eliminant els `n` últims elements del vector `current_route`, on `n` és `recently_skipped_cities`. *Nota: en cas que totes les ciutats s'hagin de saltar, s'hauran d'eliminar els* `n-1` *elements del vector i no* `n`*, ja que la ciutat inicial mai té assignat un element al vector de la ruta.*

Un cop eliminades les ciutats *innecessàries* del final de la ruta, es compara la ruta amb la ruta emmagatzemada a `best_route`. Si la ruta actual comercia una quantitat total de producte major, o si les quantitats comerciades són les mateixes però la ruta actual té menys llargada, es substitueix la `best_route` per la `current_route`. En cas que els dos valors coincideixin, es manté la `best_route` intacta. Això passa perquè, com que té prioritat la ruta de l'esquerra i a la crida recursiva es calcula primer la subruta esquerra, la subruta esquerra es calcula primer que la subruta dreta, i llavors no s'ha de modificar la millor ruta.

Ara, la postcondició es compleix. S'ha comparat la ruta actual i la millor ruta (desada a `best_route`), i la millor s'ha desat a `best_route.`

Com que es compleix la postcondició, 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.

Per tant, es comença creant una còpia profunda de la ruta actual i del vaixell. *Per motius d'eficiència, en comptes de crear una segona còpia es les instàncies de* `current_route` *i de* `test_ship` .

Com que els valors `bought_amount, sold_amount, recently_skipped_cities` i `test_ship` ja han estat actualitzats a la ciutat actual, tan sols cal modificar les dues còpies de `current_route`, afegint-los `NavStep::Left` i `NavStep::Right`, respectivament.

Finalment, per a continuar calculant totes les rutes possibles des de la posició actual, només cal cridar la pròpia funció dos cops, amb els paràmetres `bought_amount, sold_amount, recently_skipped_cities` i `best_route` sense modificar, però passant `current_location.left()` i `current_location.right()` en lloc del paràmetre `current_location`, i passant les còpies corresponents de `current_route` i `test_ship`.

Llavors, com que les crides recursives no finalitzaran fins que no es compleixi la seva respectiva postcondició, quan acabin podem assegurar que s'ha comparat `best_route` amb totes les rutes possibles que deriven del node actual, i, per tant, es compleix la postcondició, i pot finalitzar l'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);
    }
}
```