Justificació pràctica PRO2

Gerard Rius Husillos. Grup 23

Aclariments: S'assumeix que l'operació de restar iteradors (`it2-it1`) retorna la distància entre els dos iteradors.

JUSTIFICACIÓ ALGORISME ITERATIU

Definició:

```
void Valley::trade(const std::string& city1, const std::string& city2);
```

Precondició: Les dues ciutats són vàlides, existeixen a l'arbre que representa el riu.
Postcondició: Les dues ciutats han comerciat entre elles fent tots els canvis possibles tenint en comptes les regles de comerç de l'enunciat.

Funció fita: $f(it1,it2)=(c1.end()-it1)+(c2.end()-it2)$

Invariant: `c1.begin()<=it1<=c1.end()`, `c2.begin()<=it2<=c2.end()`, a més a més, les llistes sobre les que iterem estan ordenades i s'ha intentat comerciat amb les ciutats desde `c1.begin()` fins a `it1-1` i `c2.begin()` fins a `it2-1`.

Justificació:

- ➔ **Inicialització:** Abans de la primera iteració els iteradors valen `it1=c1.begin()` i `it2=c2.begin()` que compleixen l'invariant. Cap ciutat tampoc ha comerciat, que també compleix.
  ```
  auto it1 = c1.begin();
  auto it2 = c2.begin();
  ```
- ➔ **Cos:** Cada iteració intenta comerciar amb els productes `it1` i `it2` depenent de si es comercia amb el producte o no, els iterados s'incrementen d'acord amb les condicions.
- ➔ **Final:** El bucle acaba quan `it1==c1.end()` o `it2==c2.end()`, per tant, l'invariant es compleix també al final del bucle.

Codi de la funció:

```
void Valley::trade(const std::string& city1, const std::string& city2) {
   City& c1 = mCities[city1];
   City& c2 = mCities[city2];

   auto it1 = c1.begin();
   auto it2 = c2.begin();
   while (it1 != c1.end() && it2 != c2.end()) {
       if (*it1 == *it2) {
```

```
            size_t id = *it1;
            // If both cities have a surplus, there is no trade
            if (c1.surplus(id) && c2.surplus(id)) {
                it1++;
                it2++;
                continue;
            }
            if (!c1.surplus(id) && !c2.surplus(id)) {
                it1++;
                it2++;
                continue;
            }

            if (c1.surplus(id)) {
                int needs = c2.needs(id);
                int surplus_count = c1.surplusCount(id);

                int trade_count = std::min(needs, surplus_count);
                c1.addProducts(id, -trade_count);
                c2.addProducts(id, trade_count);
            } else {
                int needs = c1.needs(id);
                int surplus_count = c2.surplusCount(id);

                int trade_count = std::min(needs, surplus_count);
                c1.addProducts(id, trade_count);
                c2.addProducts(id, -trade_count);
            }

            it1++;
            it2++;
        } else if (*it1 < *it2)
            it1++;
        else
            it2++;
    }
}
```

JUSTIFICACIÓ ALGORISME RECURSIU

Precondició: Les dades de data han de ser majors de 0. root ha de ser vàlid i no buit. Les dades del vaixell també s'assumeixen correctes.
Postcondició: S'ha trobat la ruta més profitosa per al vaixell. No canvia l'estat del riu.

Cas base: Quan l'arbre és buit, retorna una ruta buida, com s'espera. Hi ha un segon cas base que es quan la barca no té més productes per vendre, però aquest cas no és necessari, ja que les condicions de comerç amb les ciutats tenen en compte aquests valors, i està per millorar rendiment.
Pas inductiu: Assumint per hipòtesi d'inducció que la funció funciona per qualsevol arbre de mida *n*, després de comerciar s'escull entre la millor opció d'entre esquerra i dreta.

Funció fita: $f(n) = n$ on *n* és la profunditat de l'arbre.

Codi de la funció:

```
std::vector<Ship::Direction> Valley::findBestRoute(
    RouteData& data, const BinTree<std::string>& root, int buy_count,
    int sell_count) {
    // 0. Base case (Reached end of tree)
    if (root.value() == "#") {
        return std::vector<Ship::Direction>();
    }

    // 0.5 Early exit case (Ship sold and bought everything)
    if (sell_count <= 0 && buy_count <= 0) {
        return std::vector<Ship::Direction>();
    }

    // 1. Calculate buy and sell quantity in current city
    City c = mCities[root.value()];  // Get copy of current city

    size_t buy_id = mShip.buy_id - 1;
    size_t sell_id = mShip.sell_id - 1;

    if (c.needs(sell_id) > 0 && sell_count > 0) {
        int trade_count = std::min(c.needs(sell_id), sell_count);
        sell_count -= trade_count;
        data.sell_count += trade_count;
    }

    if (c.surplus(buy_id) && buy_count > 0) {
        int trade_count = std::min(c.surplusCount(buy_id), buy_count);
        buy_count -= trade_count;
```

```
        data.buy_count += trade_count;
    }

    // 2. Calculate best routes from left and right
    RouteData best_left_data;
    auto best_left =
        findBestRoute(best_left_data, root.left(), buy_count, sell_count);
    RouteData best_right_data;
    auto best_right =
        findBestRoute(best_right_data, root.right(), buy_count, sell_count);

    // 2.5 If no trades have happened early return here with current trades
    if (best_left_data.total() == 0 && best_right_data.total() == 0)
        return std::vector<Ship::Direction>();

    // 3. Choose best route (left or right)
    Ship::Direction choosed_dir = Ship::Direction::None;
    if (best_right_data.total() > best_left_data.total())
        choosed_dir = Ship::Direction::Right;
    else if (best_left_data.total() > best_right_data.total())
        choosed_dir = Ship::Direction::Left;
    // Equal
    else if (best_right.size() < best_left.size())
        choosed_dir = Ship::Direction::Right;
    else
        choosed_dir = Ship::Direction::Left;

    // 4. Update and return calculated best route
    if (choosed_dir == Ship::Direction::Left) {
        data.buy_count += best_left_data.buy_count;
        data.sell_count += best_left_data.sell_count;
        best_left.push_back(choosed_dir);  // Route is defined back to front
        return best_left;
    } else {
        data.buy_count += best_right_data.buy_count;
        data.sell_count += best_right_data.sell_count;
        best_right.push_back(choosed_dir);  // Route is defined back to front
        return best_right;
    }
}
```