Specyfikacja implementacyjna
Projekt indywidualny

Łukasz Knigawka

11 listopada 2018

Spis treści

1 Wstęp

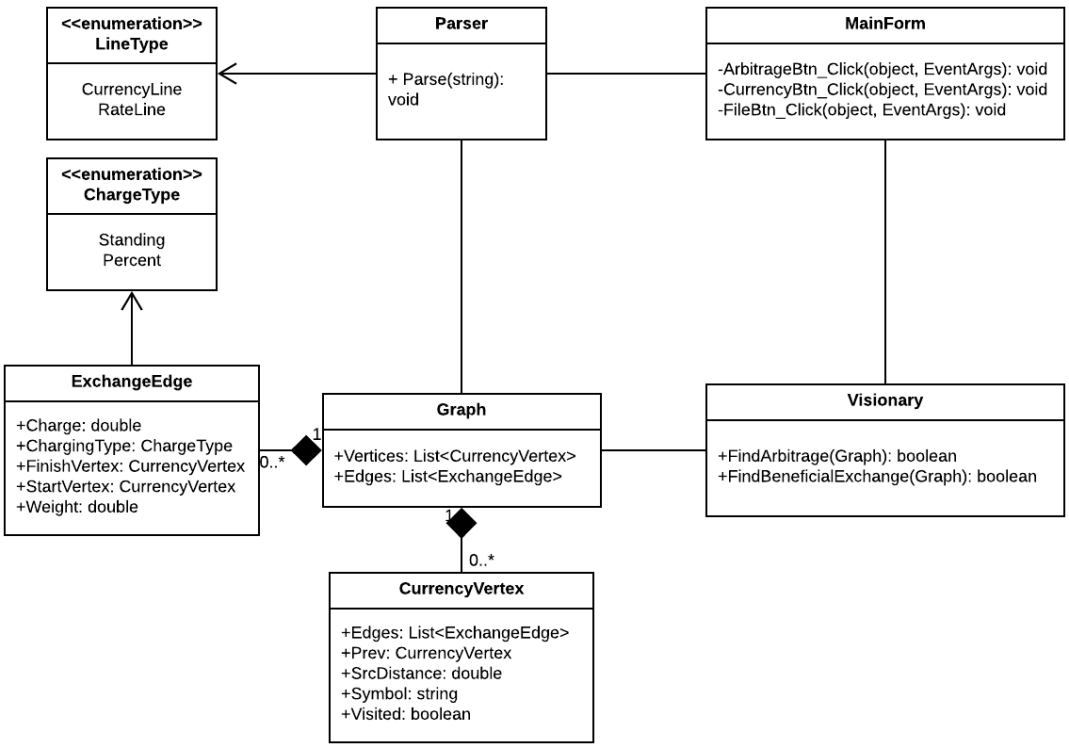
W specyfikacji funkcjonalnej opisano kluczowe dla zrozumienia problemu pojęcia: *wymiana waluty, kurs walutowy, opłata za wymianę, arbitraż.*

W specyfikacji implementacyjnej nie zostały zdefiniowane terminy leżące u podstaw informatyki, takie jak *graf, graf skierowany, algorytm, algorytm grafowy.*

W przedstawianej koncepcji implementacji rozwiązania problemu, opisane w *specyfikacji funkcjonalnej* zadanie tłumaczy się na język informatyki poprzez przedstawienie problemu na grafie. Graf ten jest skierowany, przy czym krawędzie (skierowane) istnieją jedynie między tymi wierzchołkami, dla których takie połączenie wynikać będzie z danych wejściowych. Przedstawiana sytuacja wymiany walut tylko powierzchownie ma swoje odbicie na rzeczywistym rynku wymiany walut, na czas realizacji zadania zapominamy jednak o realiach współczesnego świata.

Przy realizacji zadania wykorzystany zostanie język C# w wersji 7.3 oraz .NET framework w wersji 4.7.2. Interfejs graficzny zostanie utworzony dzięki bibliotece *WindowsForms*, która jest przestarzała i nie pozwala na wykonywanie wizualnie zadowalających projektów, lecz pozwala na tworzenie bardzo szybkich prototypów. Taki wybór należy uzasadnić tym, iż sercem programu jest algorytm, a głównym celem zadania nie jest doskonalenie tworzenia aplikacji desktopowych. Rozwiązanie zostanie utworzone, przetestowane i uruchomione na komputerze z 64-bitowym systemem Windows10, procesorem Intel Core i7-6700HQ oraz pamięcią RAM 16GB.

2 Diagram klas



Rysunek 1: Diagram klas

Przedstawiony diagram klas ma na celu przybliżenie koncepcji rozwiązania problemu. Nie zawiera w sobie oczywistych pól i metod dostępowych. Nie ukazano wszystkich relacji mię-

dzy klasami. Jako że rozwiązanie nie zostało jeszcze zaimplementowane, niektóre rozwiązania zapewne zmienią swoją formę w ostatecznej wersji projektu.

3 Opis klas

3.1 Klasy MainForm oraz Program

Klasa *Program* została pominięta na wykresie, z tego powodu że jest to typowa klasa dla architektury prostej aplikacji desktopowej. Uruchamiany jest za jej pomocą interfejs programu, znajduje się w niej metoda *Main*.
Klasa *MainForm* zawiera metody obsługujące interakcję użytkownika z interfejsem graficznym. Obsłużony jest przycisk wyboru pliku, po którego kliknięciu pojawia się okno dialogowe wyboru pliku. Rozwiązanie jest utworzone w taki sposób, iż użytkownik nie jest w stanie wybrać pliku innego niż plik tekstowy o rozszerzeniu *.txt*, gdyż tylko takie pliki są widoczne w eksploratorze. W klasie tej znajduje się też obsługa kliknięć pozostałych przycisków - odnalezienia korzystnej wymiany oraz odnalezienia arbitrażu. Pobierane są wtedy dane z wypełnionych pól tekstowych oraz opcjonalnie z pliku tekstowego, w zależności od wybranego rodzaju źródła danych wejściowych. Klasa komunikuje się z klasą przetwarzającą dane wejściowe i tworzącą graf.

3.2 Klasa Parser

Sercem tej klasy jest metoda *Parse*, która przetwarza dane wejściowe. Bardzo możliwe, że aby zachować względną prostotę i czystość kodu konieczne będzie utworzenie mniejszych metod, które będą prowadziły do odczytania danych. Dane czytane są linijka po linijce - jeśli linia zaczyna się od znaku *„#”*, to algorytm przechodzi do następnej linii. Jednocześnie, wraz z napotkaniem linii zaczynającej się od takiego znaku zwiększana jest zmienna lokalna świadcząca o spodziewanym typie linii. To znaczy, że po jednym odnotowanym wystąpieniu takiego znaku spodziewamy się linii opisujących waluty, a po dwóch wystąpieniach spodziewamy się opisu kursu walut. Dane są zbierane, a na ich podstawie budowane jest drzewo. Prawdopodobnie najtrudniejszym zadaniem jest obsłużenie błędnie sformatowanych danych.

3.3 Klasy Graph, ExchangeEdge oraz CurrencyVertex

Klasa *CurrencyVertex* symbolizuje wierzchołek grafu, który reprezentuje rodzaj waluty. Zawiera symbol waluty (trzyznakowy ciąg wielkich liter), informację czy wierzchołek był już odwiedzony (aby znaleźć korzystną wymianę walut), informację o najkrótszej ścieżce z wierzchołka źródłowego do danego wierzchołka, referencję do poprzednika oraz listę krawędzi tego wierzchołka. Zaleca się skorzystanie z *właściwości zaimplementowanych automatycznie*. Klasa *ExchangeEdge* symbolizuje wymianę walut. Oprócz wagi danej krawędzi należy zawrzeć w niej informację o rodzaju i wysokości towarzyszącej jej opłaty. Klasa *Graph* zawiera w sobie zbiór wierzchołków i krawędzi.

3.4 Klasa Visionary

Klasa *Visionary* zawiera w sobie algorytmy wyszukujące najkorzystniejszą wymianę oraz arbitraż. Z algorytmicznego punktu widzenia jest to najważniejsza klasa w projekcie.

4 Opis algorytmu

Funkcje programu zostały opisane w *specyfikacji funkcjonalnej*. Pierwsza z nich sprowadza się do znalezienia najkrótszej drogi z danego wierzchołka źródłowego do danego wierzchołka docelowego w grafie ważonym. Realizacja tego zadania będzie opierać się o odnalezienie najkrótszej ścieżki z wierzchołka źródłowego do pozostałych wierzchołków grafu. Druga z nich koncentruje się na wyszukaniu w grafie ujemnego cyklu.

W rozwiązaniu można wykorzystac algorytm *Bellmana-Forda*. Jego zaletą względem algorytmu *Dijkstry* w przypadku omawianego problemu jest możliwość występowania ujemnych wag wierzchołków. Gdybyśmy jako wierzchołki przyjęli waluty, a jako wagi krawędzi kursy walut, to nasz problem korzystnej wymiany sprowadzałby się do znalezienia najdłuższej ścieżki między walutą wymienianą i pożądaną. Natomiast jeśli wagi wierzchołków zlogarytmujemy i pomnożymy przez *-1*, otrzymujemy problem znalezienia najkrótszej ścieżki w grafie. Do otrzymania logarytmu można wykorzystać metodę *Math.Log()*, a chcąc dokonać powrotnej konwersji można wykorzystać *Math.Exp()*.

Realizacja algorytmu Bellmana-Forda przedstawiona w *„Wprowadzeniu do Algorytmów" (Cormen, Rivest, Leiserson, Stein)* została przedstawiona poniżej.

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3       for each edge (u, v) ∈ G.E
4            RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6       if v.d > u.d + w(u, v)
7            return FALSE
8  return TRUE
```

Rysunek 2: Algorytm Bellmana-Forda zapisany w pseudokodzie

W tym przypadku poprzez G oznaczony jest graf, przez V zbiór wierzchołków, przez E zbiór krawędzi, przez w funkcja wagowa, przez d oszacowanie wagi najkrótszej ścieżki, przez s źródło (*source*), a przez u i v wierzchołki. Algorytm korzysta z *relaksacji* krawędzi, czyli ze sprawdzenia, czy przechodząc przez wierzchołek u można znaleźć krótszą od dotychczas najkrótszej ścieżki do v. Jeśli to możliwe, aktualizowana jest wartość najkrótszej ścieżki do danego wierzchołka i aktualizowany jest poprzednik wierzchołka. Algorytm zwraca *TRUE*, gdy graf nie zawiera cykli o ujemnych wagach (osiągalnych ze źródła). W przeciwnym wypadku zwraca *FALSE*. Jest to więc perfekcyjny algorytm do wyznaczania arbitrażu, gdyż ten występuje gdy graf zawiera cykl ujemny.

Utrudnieniem w realizacji zadania są dodatkowe opłaty za wykonanie każdej wymiany walutowej, jednak trzeba je po prostu uwzględnić przy szukaniu ścieżki w grafie, przy każdym porównaniu obecnej wagi drogi od źródła do konkretnego wierzchołka z alternatywą.

5 Testy

Zakłada się użycie przy tworzeniu testów jednostkowych *Visual Studio Unit Testing Framework*. Narzędzie to znane jest także pod nazwą *MSTest*. Oprócz testów jednostkowych zostanie przeprowadzone seria testów manualnych. Testy powinny według założeń powstawać w czasie pisania kodu programu. Większość testów jednostkowych będzie imitować testy manualne.