Specyfikacja implementacyjna
programu *"grapher"*

Szymon Półtorak i Sebastian Sikorski

28.04.2022r

Streszczenie

Niniejszy dokument omawia tematykę projekt pod kątem jego implementacji. Tłumaczymy w niej w jaki sposób podeszliśmy do budowy oprogramowania i metodykę przeprowadzania testów.

Spis treści

1 Cel Projektu - streszczenie

Celem projektu było stworzenie programu mającego za zadanie generowanie grafów, sprawdzanie ich spójności oraz wyszukiwanie w nich najkrótszej ścieżki między zadanymi przez użytkownika punktami. Grafi są typu *kartka w kratkę*.

- Wage Mode – program generuje graf o losowych wagach dróg między wierzchołkami w taki sposób, że jest on spójny,
- Edge Mode – program losuje istnienie krawędzi między wierzchołkami grafu oraz wagi do momentu powstania grafu spójnego. Do sprawdzania wykorzystuje algorytm BFS,
- Random Mode – program losuje wagi dróg oraz krawędzie między wierzchołkami. W tym trybie graf może być niespójny,
- Read Mode – program odczytuje odpowiednio sformatowany plik i szuka najkrótszej ścieżki między podanymi przez użytkownika punktami za pomocą algorytmu Dijkstry.

Po szczegóły dotyczące tematyki projektu odsyłamy do specyfikacji funkcjonalnej.

2 Środowisko powstawania programu

Podczas tworzenia programu kluczowe są odpowiednio dobrane narzędzia oraz stałość ich wersji podczas pracy. Narzędzia te prezentujemy w tabeli poniżej:

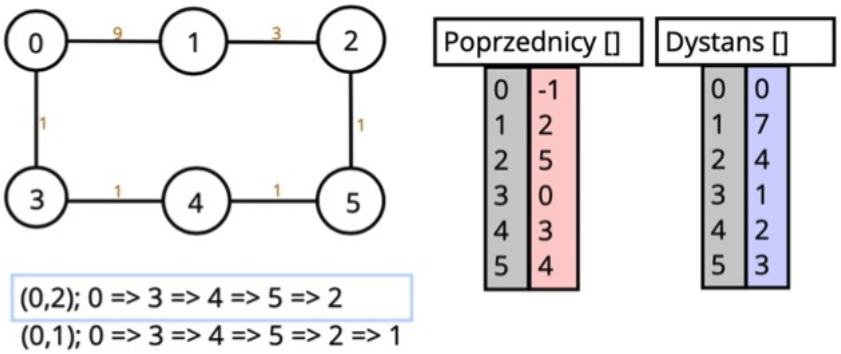
Nazwa	Wersja
Java	Java SE 17
Java Development Kit	17.0.3.
Apache Maven	3.8.5.
InteliJ IDEA	2021.3.3
Git	2.30.2.
JavaFX	18.0.1.

3 Algorytmy

Nasz program wykorzystuje dwa algorytmy, które opisujemy w poniższych podrozdziałach.

3.1 Algorytm Dijkstry

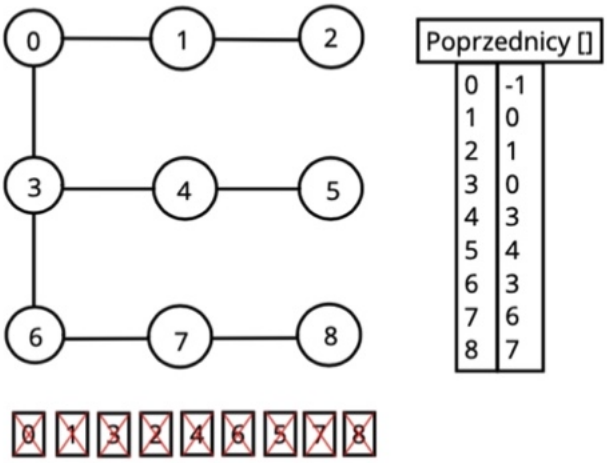
Algorytm Dijkstry liczy najkrótszą odległość od wierzchołka początkowego do wszystkich innych wierzchołków, ale w naszej implementacji skupiamy się jedynie na najkrótszej ścieżce między wierzchołkami zadanymi przez użytkownika. Algorytm ten korzysta z kopca pełniącego role kolejki priorytetowej oraz trzech tablic przechowujących poprzedników, wagi połączeń i i całowity dystans. Algorytm dodaje odwiedzane wierchołki do kolejki priorytetowej a nastepnie pobiera je z niej akutalizując dystans dopóki kopiec nie jest pusty. Następnie zaczynając od wierzchołka końcowego (podanego przez użytkownika) cofamy się aż trafimy do wierzchołka początkowego. Podczas cofania zapisujemy przez jakie wierzchołki przeszliśmy oraz jaka była waga takiego przejścia.



Rysunek 1: Przykładowe działanie algorytmu Dijksty.

3.2 Breadth-first search(BFS)

Nasza implementacji algorytmu BFS opiera się na sprawdzaniu tak zwanej "silnej spójności", algorytm ten wywoływany jest z każdego wierchołka, rozpoczynając od wierchołka pierwszego. Algorytm w celu sprawdzenia spójności tworzy tablicę poprzedników o długości odpowiadającej ilości wierzchołków oraz zapełnia ją wartościami -1. Rozpoczynając iteracje od wierzchołka zero aż do ostatniego wierzchołka. Algorytm sprawdza z jakimi wierzchołkami połączony jest obecny i dopisuje te wierzchołki do kolejki typu FIFO, czyli kolejka typu first in first out, czyli pierwszy wchodzi i wychodzi, jednocześnie uzupełniając tablicę poprzedników. Po takim kroku algorytm przechodzi do kolejnego wierzchołka pobierając jego ID z kolejki. Przy dotarciu do końca kolejki algorytm musi sprawdzić czy jednym wierzchołkiem bez poprzednika jest wierzchołek zero, w przeciwnym przypadku test spójności jest negatywny.



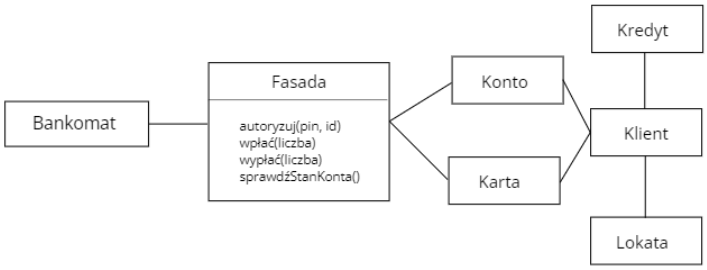
Rysunek 2: Przykładowe działanie algorytmu BFS.

4 Budowa programu

W tym rozdzialne omówimy zagadnienie wzorców projektowych, budowę oraz diagram klas naszego programu.

4.1 Wybrany wzorzec projektowy

Niniejszy projekt oparty jest na wzorcu projektowym fasady. Powoduje to stworzenie jednego prostego interfejsu służącego do sterowania programem a jego dodatkową zaletą jest ukrycie przed użytkownikiem złożoności programu.



Rysunek 3: Przykładowe zastosowanie fasady na bazie bankomatu.

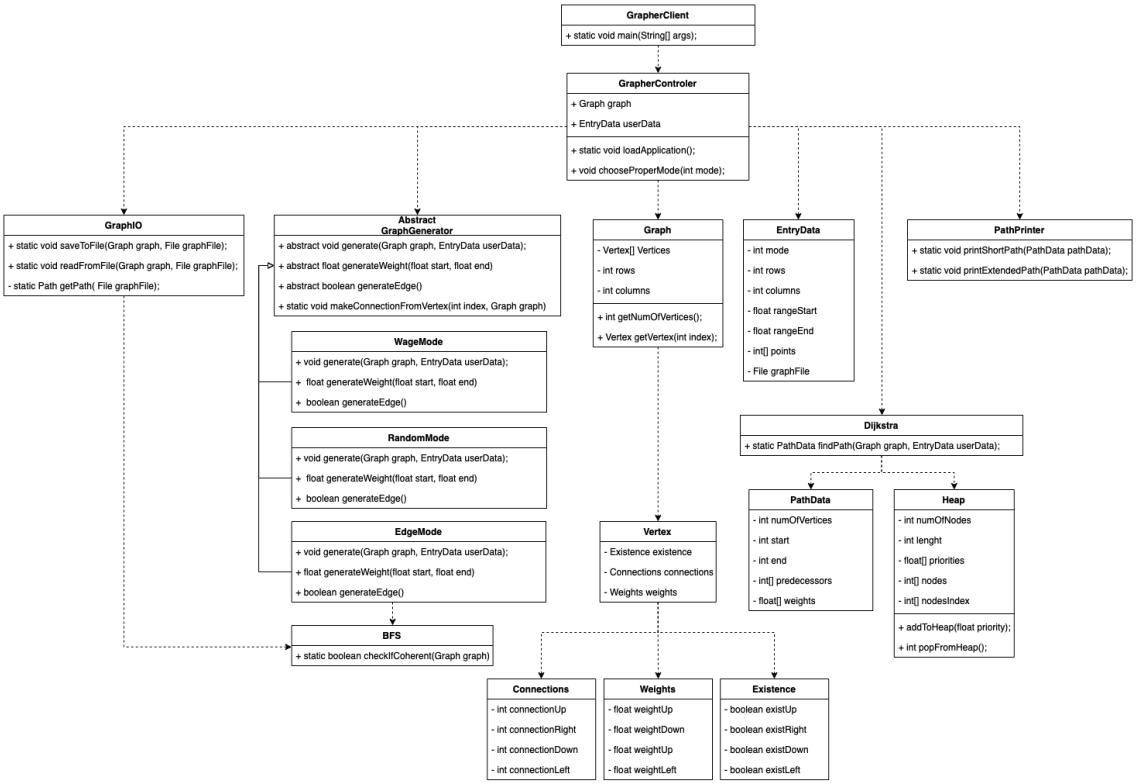
4.2 Wykorzystane klasy

Poniższa lista zawiera klasy wykorzystane w programie. Ich powiązania widoczne są na diagramie klas w następnym podrozdzialne.

- `GrapherClient` – odpowiedzialna za interakcje z użytkownikiem,
- `GrapherControler` – klasa będąca fasadą, ukrywa złożoność aplikacji przed użytkownikiem,
- `EntryData` – klasa odpowiedzialna za przechowywanie informacji wprowadzonych przez użytkownika,
- `Graph` – klasa przechowująca graf,
- `Vertex` – klasa przechowująca wierchołek oraz informacje na temat krawędzi wychodzących z danego wierchołka,
- `Existence` – klasa przechowująca informacje o tym, czy istnieje połaczenie z danego wierchołka w danym kierunku,
- `Weights` – klasa przechowująca informacje o wagach połączeń między punktami,
- `Connect` – klasa przechowująca informacje o tym dokąd prowadzi dane połączenia,
- `GraphIO` – klasa odpowiedzialna za obsługę plików,
- `GraphGenerator` – klasa odpowiedzialna za generację grafu,
- `WageMode` – klasa odpowiedzialna za tryb wag,
- `EdgeMode` – klasa odpowiedzialna za tryb krawędzi,
- `RandomMode` – klasa odpowiedzialna za tryb losowy,
- `Dijkstra` – klasa odpowiedzialna za wyszukiwanie najkrótszej ścieżki,
- `PathPrinter` – klasa odpowiedzilna za wyświetlanie obliczonej już najkrótszej ścieżki,
- `BFS` – klasa odpowiedzialna za sprawdzanie spójności grafu,
- `Heap` – klasa odpowiedzialna za kolejkę priorytetą w postaci kopca,
- `PathData` – klasa odpowiedzialna za przechowywanie informacji klucziowych dla wyszukiwania najkrótszej ścieżki.

4.3 Diagram klas

Dokładne powiązania między klasami reprezentuje poniższy diagram. Zawiera on tylko najważniejsze metody każdej klasy.



Rysunek 4: Diagram klas.

5 Testowanie programu

Testowanie jednostkowe programu będzie przeprowadzone z wykorzystaniem framework'u JUnit oraz biblioteki AssertJ. Testowane będą najważniejsze funkcjonalności programu oraz poszczególne interakcje między nimi. Testom poddana będzie także umiejętność rozpoznawania przez program niepoprawnych danych wejściowcyh tak aby program zachowywał się zgodnie z implementacją funkcjonalną.

6 Wprowadzanie zmian

Zmiany wprowadzane były za pomocą systemu kontroli `Git`. Poszczególne zadania były wykonywane za pomocą osobnych gałęzi (branchy), a zmiany wprowadzane za pomocą commitów. Złączanie gałęzi z główną *masterem* było przeprowadzane po akcpetacji kodu przez drugiego członka zespołu.

7 Konwencje nazewnicze

W celu utrzymania czytelności i przejrzystości kodu przyjęliśmy niniejsze konwencje nazewnicze:

- Wszystkie nazwy i komunikaty w języku angielskim,
- Wszelkie instrukcje wewnątrz instrukcji `if` i wszystkich pętli muszą być wewnątrz nawiasów klamrowych,
- Adnotacje znajdują się zawsze w wierszu poprzedzającym metodą, do której się odnoszą,
- Nazwy zmiennych i klas są pisane w konwencji CamelCase. Zmienne zaczynam małą literą i rozpoczęcie drugiego słowa zaczyna się wielką literą, czyli słowo "is empty" jako zmienna i metoda to "isEmpty", a klasa "IsEmpty".