Specyfikacja implementacyjna projektu pt. „Patients Transport Center”

Wykonali: Antoni Malinowski, Franciszek Wysocki, Bartosz Zakrzewski
Sprawdzający: mgr inż. Paweł Zawadzki
Data: 17-12-2020

Spis treści

1 Cel dokumentu

Niniejszy dokument przedstawia plan implementacji programu, który zarządza transportem pacjentów do szpitali. Zostaną w nim podane szczegóły dotyczące środowisk deweloperskich członków zespołu, zasad pracy z systemem kontroli wersji, rodzajów testów, struktury projektu, struktur danych i algorytmów.

2 Cel projektu

Celem projektu jest zapewnienie optymalnego transportu pacjentów z obszaru zdefiniowanego przez najmniejszy zbiór wypukły zawierający w sobie wszystkie obiekty. Program powinien decydować czy dany pacjent znajduje się w tym obszarze, transportować go do najbliższego szpitala i w razie braku wolnych łóżek znajdować szpital, do którego czas transportu jest najkrótszy.

3 Podstawowe informacje

- aplikacja będzie posiadała graficzny interfejs użytkownika, który pozwoli między innymi na dodanie pacjenta na mapie, wczytanie pliku, zmianę tempa;
- aplikacja będzie wielowątkowa – zostanie oddzielony wątek wykonujący obliczenia od wątku wyświetlającego symulację.

4 Środowisko deweloperskie

Program będzie implementowany przez trzech członków zespołu na ich indywidualnych komputerach.

Aplikacja zostanie napisana w języku programowania Java w wersji 13.0.2 (JDK).

Każdy członek zespołu będzie pracował na systemie Windows 10 i na tych samych wersjach narzędzi deweloperskich:

- IDE - Intellij 2020.2.4 Community Edition;
- Apache Maven w wersji 3.6.3;
- Git w wersji 2.29.2;
- Projektor w wersji 2.3.1.

Pozostałe narzędzia:

- Sprite'y: rysunek karetki, szpitali, obiektów (pomników), dróg, mapy, pacjentów będą stworzone w programie Paint, Paint 3D i GIMP 2;
- Kontakt między członkami zespołu będzie odbywał się za pomocą aplikacji Messenger i Microsoft Teams;
- Udostępnianie plików dla wszystkich członków zespołu nastąpi za pomocą aplikacji OneDrive;
- W podziale obowiązków i zauważeniu progresu pomoże nam Tablica Kanban na stronie Trello.

Wykorzystane pakiety i biblioteki:

- Do stworzenia GUI zostaną użyte pakiety java.awt oraz javax.swing;
- Do napisania testów zostaną wykorzystane biblioteki JUnit 5, AssertJ oraz Mockito.

5 Zasady wersjonowania

Wersjonowanie odbędzie się za pomocą systemu kontroli wersji – git. Commity (oraz kod) będą pisane w języku angielskim.

Commity mogą zostać otagowane, jeżeli zajdzie taka potrzeba (np. tagiem „FINAL_RELEASE_GR3_1", „PROOF_OF_CONCEPT" czy „STABLE_01").

Praca z systemem kontroli wersji git będzie rozłożona na wiele gałęzi. Łączenie ich będzie wykonywane za pomocą komendy git merge.

Gałęzie będą nazywane za pomocą „kebab case" np. „data-verification", „graph-representation".

6 Testowanie

Weryfikacja poprawności działania programu odbędzie się przy pomocy testów jednostkowych.

Do ich napisania zostaną wykorzystane biblioteki: JUnit 5, AssertJ oraz Mockito.

Testy zostaną stworzone zgodnie z zasadą F.I.R.S.T. i będą sprawdzały poprawność działania konkretnych metod publicznych ze szczególnym naciskiem na warunki brzegowe, odporność na niepożądane warunki (takie jak np. podanie niezainicjowanej zmiennej jako argument funkcji).

Nazwy testów będą pisane zgodnie z konwencją:

```
should_expectedBehavior_when_stateUnderTest
```

Natomiast ciało funkcji testujących z konwencją given/when/then.

Testy będą automatycznie uruchamiane za pomocą Mavena przed stworzeniem pliku JAR.

7 Struktura projektu

Nazwa projektu to „patients-transport-center".
Projekt z racji, że będzie posiał graficzny interfejs użytkownika będzie podzielony na dwa główne pakiety:

- **"pl.group2.optimizer.gui"** zawierający pakiety i klasy związane z java.awt i javax.swing;
- **„pl.group2.optimizer.impl"** zawierający część odpowiadającą za obliczenia.

Pakiet **pl.group2.optimizer.impl** dzieli się na:

- **„pl.group2.optimizer.impl.algorithms"** zawierający klasy reprezentujące algorytmy;
- **„pl.group2.optimizer.impl.io"** zawierający klasy obsługującą wczytywanie z pliku i obsługę błędów;
- **„pl.group2.optimizer.impl.items"** zawierający klasy reprezentujące szpitale, obiekty, pacjentów, drogi oraz skrzyżowania;
- **„pl.group2.optimizer.impl.structures"** zawierający klasy reprezentujące graf i inne struktury danych.

8 Klasy

Klasy czy pakiety mogą wymagać zmiany lub utworzenia nowych, dlatego członkowie zespołu powinny pozostawać w kontakcie.

Klasa główna „**Optimizer**" (zawierająca metodę main()) odpowiadająca za sterowanie aplikacją, będzie znajdować się w pakiecie **pl.group2.optimizer**.

Klasy odpowiadające za algorytm i optymalizację:

1. Klasy pakietu **„pl.group2.optimizer.impl.algorithms"**:
   - **AreaHandledByAmbulance**: jest to klasa reprezentująca obszar wyznaczony przez szpitale oraz obiekty graniczne,w którym program poszukuje pacjentów do obsłużenia. Posiada metodę **contains()**, która zawiera implementacje algorytmu **Ray casting** i służy do wykrywania, czy pacjent znajduje się w wyznaczonym obszarze;
   - **GrahamAlgorithm**: zawiera implementacje algorytmu Grahama, który służy do wyznaczenia granic terytorium, w którym karetka jest w stanie dotrzeć do pacjenta;
   - **DijkstraAlgorithm**: klasa służąca do wyznaczania takiej drogi pomiędzy szpitalami (po której karetka transportująca pacjenta będzie musiała przejechać) aby była ona najkrótsza z możliwych;
   - **ShortestDistanceChecker**: zajmuje się szukaniem najbliższego szpitala z dowolnego miejsca (które jest w istocie miejscem w którym "pojawia się" pacjent) znajdującego się na terytorium obsługiwanym przez karetki;
   - **IntersectionSearcher**: klasa ta zajmuje się wyszukiwaniem przecięć pomiędzy gałęziami grafu łączącego szpitale i dodawaniem nowo powstałych dróg do listy wszystkich dróg pomiędzy szpitalami.

2. Klasy pakietu **„pl.group2.optimizer.impl.io”**:

   - **TextFileReader**: zajmuje się czytaniem plików wejściowych i zapisywaniem zawartych w nich danych do odpowiednich struktur w programie.
     Dodatkowo sprawdza czy plik wejściowy jest poprawny. Jeżeli nie to wywołuje metody z klasy ErrorHandler;
   - **ErrorHandler**: zawiera metody, które wyświetlają odpowiedni komunikat o tym jaki błąd się pojawił.
     np. błąd formatowania w pliku, i w której linii się on znajduje (aby użytkownik mógł poprawić go i załadować dane ponownie).
     Jeżeli zajdzie taka potrzeba to program zakończy się z odpowiednim kodem błędu.

3. Klasy pakietu **„pl.group2.optimizer.impl.items”**:

   - **Hospital**: klasa służąca do tworzenia obiektów reprezentujących szpitale i przechowujących informację o nich;
   - **Hospitals**: klasa przechowująca operacje, które możemy wykonywać na liście liniowej szpitali;
   - **SpecialObject**: klasa służąca do tworzenia obiektów reprezentujących obiekty dodatkowe (np. pomniki) i przechowujących informację o nich;
   - **Path**: klasa służąca do tworzenia obiektów reprezentujących drogi/ścieżki między szpitalami i przechowujących informację o nich;
   - **Paths**: klasa posiadająca operacje, które można wykonać na drogach (np. sprawdź czy drogi się nie przecinają);
   - **Patient**: klasa służąca do tworzenia obiektów reprezentujących pacjentów i przechowujących informację o nich;
   - **Patients**: klasa posiadająca operacje, które można wykonywać na liście pacjentów (np. dodaj pacjenta, sprawdź czy leży poza obszarem);
   - **Intersection**: klasa reprezentująca skrzyżowanie, jest ona potrzebna, aby algorytm poszukujący najkrótszej ścieżki mógł przecięcie dróg jako wierzchołek w drodze do szpitala;
   - **Intersections**: klasa przechowująca operacje, które możemy wykonywać na większej ilości skrzyżowań;
   - **Vertex**: interfejs, implementowany przez klasy Intersection oraz Hospital. Reprezentuje on wierzchołek grafu;

4. Klasy pakietu **„pl.group2.optimizer.impl.structures”**:
   - **Graph**: służy do stworzenia grafu, którego wierzchołki to szpitale bądź skrzyżowania, a gałęzie to drogi pomiędzy wybranymi szpitalami/skrzyżowaniami; (pacjent ma zostać przetransportowany do szpitala więc karetka „nie zatrzyma się” na skrzyżowaniu);
   - **Stack**: zawiera implementacje stosu z dodatkową metodą zwracającą również przedostatni element leżący na stosie;
   - **BinaryHeap**: zawiera implementacje minimalnego kopca binarnego wykorzystywanego w klasie MinimumPriority-Queue;
   - **MinimumPriorityQueue**: tworzy kolejkę priorytetową, której priorytetem będzie aktualnie wyliczona odległość od wierzchołka grafu;
   - **FifoQueue**: kolejka first in, first out, która posłuży do określania kolejności transportu pacjentów.

9 Algorytmy, które zostaną wykorzystane i ich opis

Implementacja poniższych algorytmów będzie oparta na pseudokodzie i opisie algorytmu, które można znaleźć:

- w książce pt. „Wprowadzenie do algorytmów" autorstwa: Clifford Stein, Ron Rivest i Thomas H. Cormen;
- na polskiej i angielskiej wikipedii;
- na innych stronach: np. geeksforgeeks.org, stackoverflow.com.

1. **Algorytm Grahama**

   Jest to efektywny algorytm znajdowania otoczki wypukłej skończonego zbioru punktów płaszczyzny.

   Zostanie on wykorzystany do zakreślenia obszaru, w obrębie którego pacjent będzie przetransportowany do szpitali.

   Rozpatrywanymi punktami będą współrzędne wszystkich obiektów.

   Złożoność obliczeniowa algorytmu: O(n * log n), gdzie n - liczba punktów.

2. **Algorytm sprawdzania czy istnieją przecinające się odcinki**

   Algorytm ten posłuży do szukania skrzyżowań dróg. Po znalezieniu przecinających się dróg zostanie utworzony obiekt reprezentujący skrzyżowanie.

   Skrzyżowanie zawiera informację w jakim punkcie (x,y) znajduje się przecięcie dróg, dzięki czemu będzie można obliczyć odległości dróg, które łączą szpitale ze skrzyżowaniem.

   Nowo utworzone drogi (szpital - skrzyżowanie) zostaną dodane do kolekcji dróg, natomiast krzyżujące się odcinki (szpital - szpital) zostaną z niej usunięte.

   Złożoność obliczeniowa algorytmu: O(n * log n), gdzie n - liczba odcinków.

3. **Algorytm Ray Casting**

   Jest to algorytm stwierdzający czy dany punkt (pacjent) znajduje się w obszarze (znalezionej wcześniej otoczce wypukłej).

   Jeżeli tak to dany pacjent zostanie przetransportowany do szpitala (lub kolejnych szpitali w przypadku braku wolnych łóżek).

   Jego implementacja znajdzie się w metodzie contains() klasy wielokąta reprezentującego obsługiwany obszar.

Złożoność obliczeniowa algorytmu: O(n^2), gdzie n - liczba wierzchołków reprezentujących otoczkę wypukłą.

4. **Algorytm znajdowania pary najmniej odległych punktów**

   Algorytm ten posłuży podczas szukania najbliższego szpitala z dowolnego miejsca należącego do obsługiwanego obszaru.

   Będzie on używany za każdym razem, gdy zostanie dodany nowy pacjent. Karetka transportuje go prosto do szpitala (nie musi jechać po drodze).

   Domyślnie złożoność obliczeniowa tego algorytmu wynosi: O(n * log n), gdzie n - liczba punktów w zbiorze (liczba szpitali). Do potrzeb projektu, algorytm nie będzie znajdował pary najmniej odległych szpitali w całym zbiorze, a jedynie najmniejszą odległość między konkretnym punktem (pacjentem), a szpitalami, dzięki czemu złożoność wynosi O(n), gdzie n - liczba punktów (szpitali).

5. **Algorytm Dijkstry**

   Jest to algorytm znajdowania najkrótszej ścieżki (ścieżki o najmniejszej sumie wag poszczególnych krawędzi - w tym przypadku wagi to miary odległości drogi).

   Posłuży on do szukania szpitala i drogi (dróg), która do niego doprowadzi.

   Gdy w odwiedzonym szpitalu nie będzie wolnych łóżek, to algorytm zostanie wykonany ponownie.

   Złożoność obliczeniowa algorytmu: O(E * log V), gdzie E - liczba krawędzi; V - liczba wierzchołków.

10 Struktury danych i abstrakcyjne typy danych

1. **Stos** - zostanie wykorzystany podczas implementacji algorytmu Grahama (będzie przechował punkty - współrzędne obiektów). Stos poza podstawowymi metodami będzie posiadał dodatkowo metodę nextToTop() - zwracającą przedostatni element;
2. **Kolejka Priorytetowa** typu min - zostanie stworzona na podstawie kopca binarnego i będzie wykorzystana do zaimplementowania algorytmu Dijkstry. Będzie przechowywała wierzchołki grafu (obiekty). Priorytetem kolejki będzie aktualnie wyliczona odległość od wierzchołka źródłowego;
3. **Kolejka FIFO** - zostanie stworzona w oparciu o listę liniową. Będzie przechować pacjentów w kolejności dodania, aby umożliwić obsługę pacjentów w określonej sekwencji;
4. **Kopiec binarny** (minimalny) - będzie przechowywał elementy w taki sposób, że rodzic będzie mniejszy lub równy swojemu dziecku. Jest to spowodowane tym, że wyciągane elementy będą w kolejności od najmniejszego do największego, gdyż tego będzie wymagała kolejka;
5. **Graf** - jego wierzchołkami będą szpitale bądź skrzyżowania, a wartościami krawędzi będą czasy potrzebne do przebycia konkretnej drogi. Zostanie on zaimplementowany za pomocą tablicy dwuwymiarowej;
6. **Tablice** - będą przechowywały elementy, kiedy ich liczba będzie stała podczas działania programu (będą to np. sprite'y szpitali). Zostaną również stworzone tablice dwumiarowe, które będą reprezentowały macierze (przechowujące wagi grafu - wartości na drogach);
7. **Listy liniowe** - będą przechowywać elementy, których liczba może zmieniać w trakcie działania programu oraz po których wymagana będzie szybka iteracja (np. lista szpitali).

11 Źródła

Aby opisać algorytmy i dowiedzieć się o potrzebnych strukturach danych i abstrakcyjnych typach danych skorzystaliśmy z:

- „Wprowadzenie do algorytmów" autorstwa: Clifford Stein, Ron Rivest i Thomas H. Cormen;
- Wikipedia.

Ten dokument został stworzony za pomocą strony overleaf.com.