# **Specyfikacja implementacyjna**
programu *"grapher"*

Szymon Półtorak, Sebastian Sikorski

02.06.2022r

**Streszczenie**

Niniejszy dokument stanowi sprawozdanie z projektu *grapher* napisanego w języku *Java*. Przedstawiamy cel projektu, użyte algorytmy, strukturę folderów oraz działanie naszego programu. Podsumowujemy projekt, współpracę i wyciągamy z niego wnioski.

# Spis treści

# 1 Cel Projektu

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.

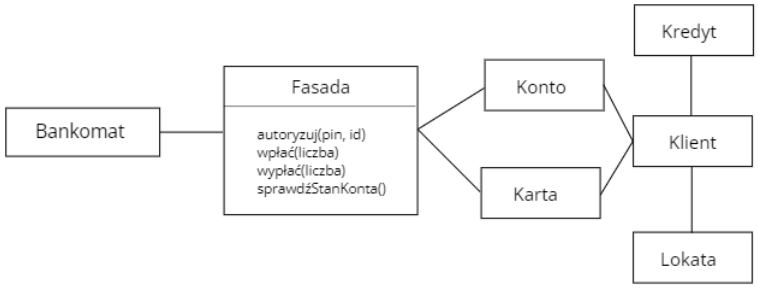
# 2 Środowisko powstawania projektu

Poniżej przedstawiamy środowisko powstania programu, czyli wykorzystane przez nas technologie.

| Nazwa | Wersja |
|---|---|
| IntelliJ Idea | 2022.1.1 |
| Apache Maven | 3.8.1 |
| JavaFX | 18.0.1 |
| JUnit | 5.8.2 |
| Java Development Kit | 17.03 LTS |
| Java Language Level | 11 |
| Git | 2.30.2. |

# 3 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 1: Przykładowe zastosowanie fasady na bazie bankomatu.

# 4 Format pliku z grafem

Program do działania w trybie Read Mode przyjmuje plik o określonych właściwościach:

- W pierwszym wierszu pliku znajduje się informacja o liczbie wierszy i kolumn jakie składają się na graf,
- W każdym następnym wierszu znajduję się informacja o tym z jakimi innymi wierzchołkami połączony jest dany wierzchołek oraz waga jaka odpowiada temu połączeniu.

Ze względu na numerowanie wierzchołków od zera, numer wiersza odpowiada numerowi wierzchołka zwiększonego o jeden. Przykładowa zawartość pliku:

```
3 3
     1 :0.33  3 :2.32
     2 :3.21
     1 :5.11  5 :2.46
     0 :0.89  4 :3.23  6 :2.21
     1 :1.23  3 :3.27  5 :2.25 7:5.12
     4 :2.33
     3 :3.63  7 :1.22
     6 :6.21  4 :1.34
     5 :4.26  7 :8.1
```

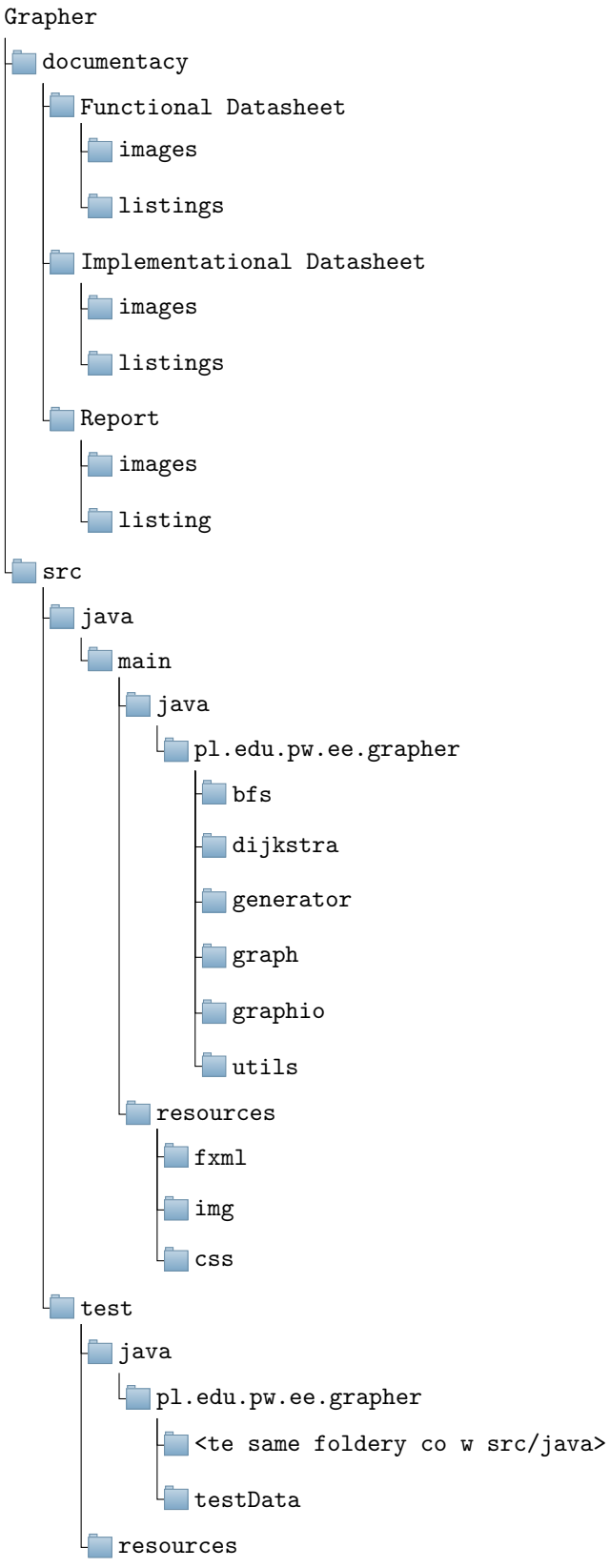
# 5 Uruchomienie Programu

# 6 Struktura Programu

W tym rozdziale przedstawiamy strukturę katalogów naszego programu oraz diagram klas.

## 6.1 Struktura folderów

Struktura różni się lekko od tej zaprezentowanej w specyfikacjach z racji przeniesienia projektu na osobną stronę z system kontroli wersji. Wszystkie nazwy zostały zmienione na język angielski.


Grapher
documentacy
Functional Datasheet
images
listings
Implementational Datasheet
images
listings
Report
images
listing
src
java
main
java
pl.edu.pw.ee.grapher
bfs
dijkstra
generator
graph
graphio
utils
resources
fxml
img
css
test
java
pl.edu.pw.ee.grapher
<te same foldery co w src/java>
testData
resources

### 6.2 Diagram Klas

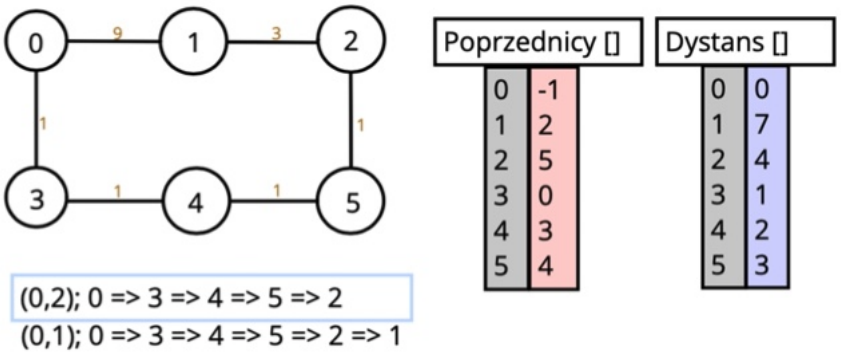
Nowy diagram klas został umieszczony w pliku HTML ze względu na jego rozbudowaną strukturę.

## 7 Wykorzystane algorytmy

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

### 7.1 Algorytm Dijkstry

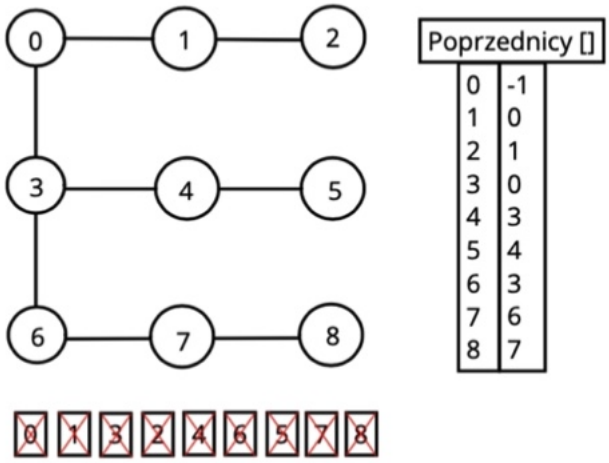
Algorytm Dijkstry liczy najkrótszą odległość od wierzchołka początkowego do wszystkich innych wierzchołków, ale w naszej implementacji skupiamy się jedynie na najkrótszej ścieżce między wierzchołkami zadanymi przez użytkownika. Algorytm ten korzysta z kopca pełniącego role kolejki priorytetowej oraz 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 2: Przykładowe działanie algorytmu Dijksty.

## 7.2 Breadth-first search(BFS)

Nasz program wykorzystuje do sprawdzania spójności algorytm BFS. Nasza implementacja różni się od pierwotnie zakładanej w specyfikacji implementacyjnej projektu, ponieważ wykorzystaliśmy algorytm *Kosaraju*, który wjaśnimy niżej. 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 BFS polega na sprawdzeniu spójności przez przechodzenie po sąsiadach danego wierzchołka i jeżeli algorytmowi uda się przejść po wszystkich wierzchołkach(oczywiście jeżeli algorytm zostanie wykonany ze wszystkich wierzchołków co gwarantuje tak zwaną *silną spóność*), czyli w tablicy odwiedzonych wierzchołków wszystkie mają wartość `true`, to znaczy, że graf jest spójny.



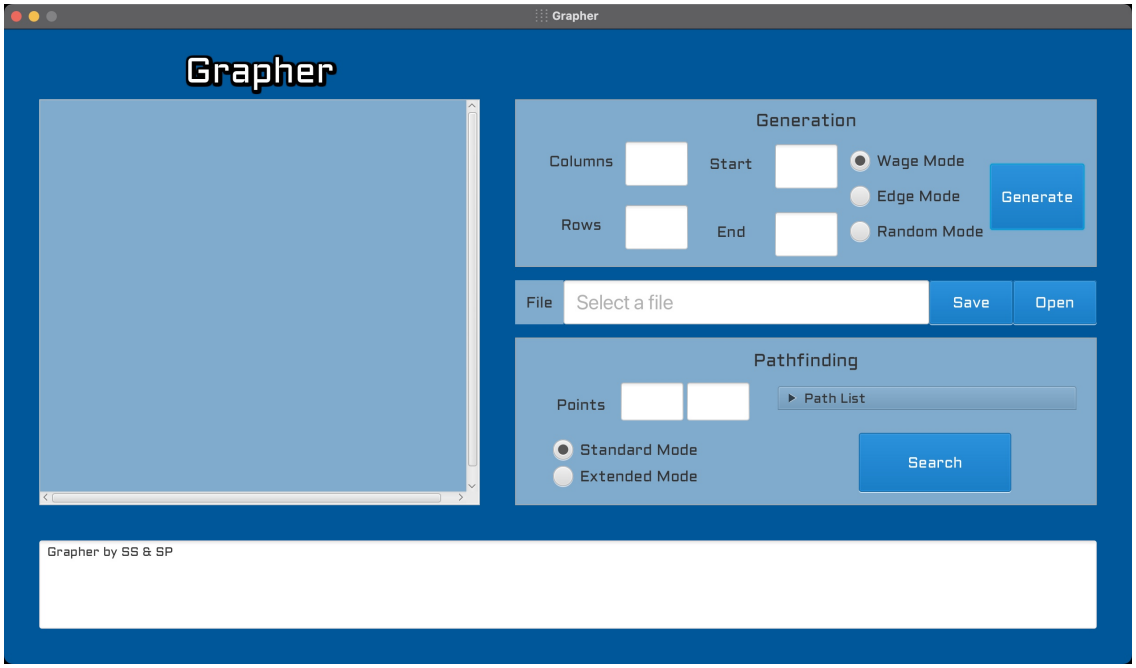

Rysunek 3: Przykładowe działanie algorytmu BFS.

## 7.3 Strongly Connected Components (Algorytm Kosaraju)

Algorytm Kosaraju polega na wykorzystaniu silnych związków pomiędzy wierzchołkami grafu. Pierwszym krokiem jest włączenie BFS'a z zerowego wierzchołka i sprawdzenie, czy graf jest spójny. Jeżeli jest spójny to trzeba odwrócić graf, to znaczy zamienić zwroty wszystkich krawędzi grafu żeby biegły w przeciwną stronę, a następnie uruchamiamy BFS'a po raz drugi ale już na odwróconym grafie. Spójność po drugim uruchomieniu sprawdzania za pomocą algorytmu zapewnia nam spójność całego grafu.

# 8 Wywołania programu

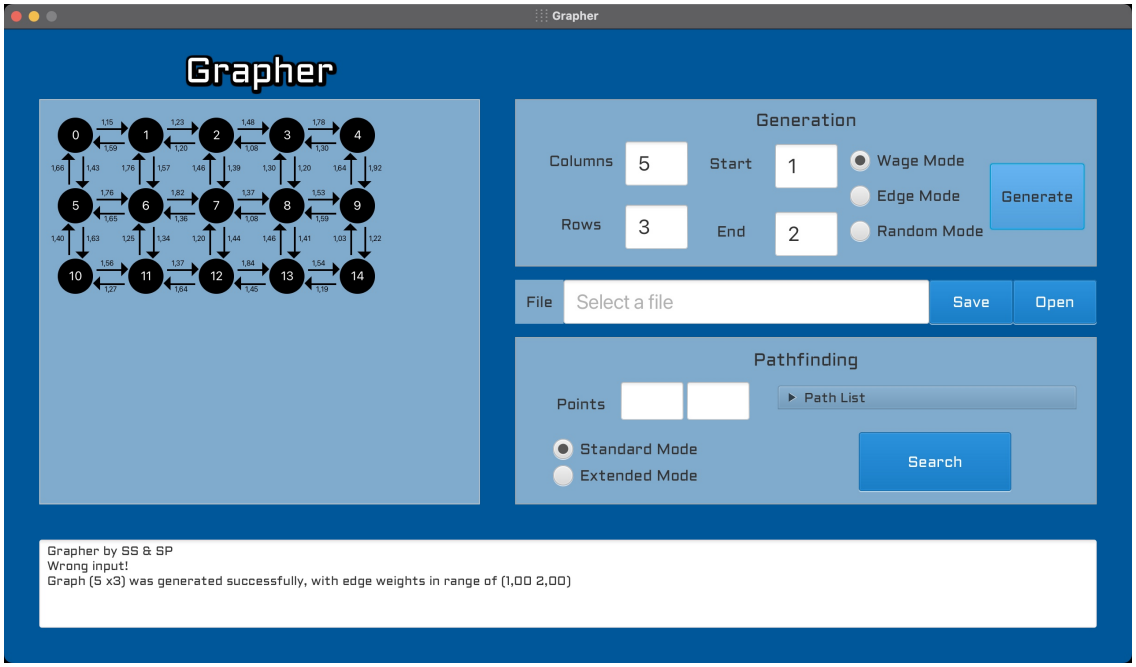
W poniższym rozdziale zostaną pokazane dwa przykładowe wywołania programu, w trybie WageMode a także EdgeMode. Zaprezentowana zostanie generacja, szukanie najkrótszej ścieżki między dwoma zadanymi punktami, a także widok listy pozwalający na szybkie przełączenie się między wcześniej wyznaczonymi ścieżkami.




Rysunek 4: Ekran startowy aplikacji witający użytkownika.

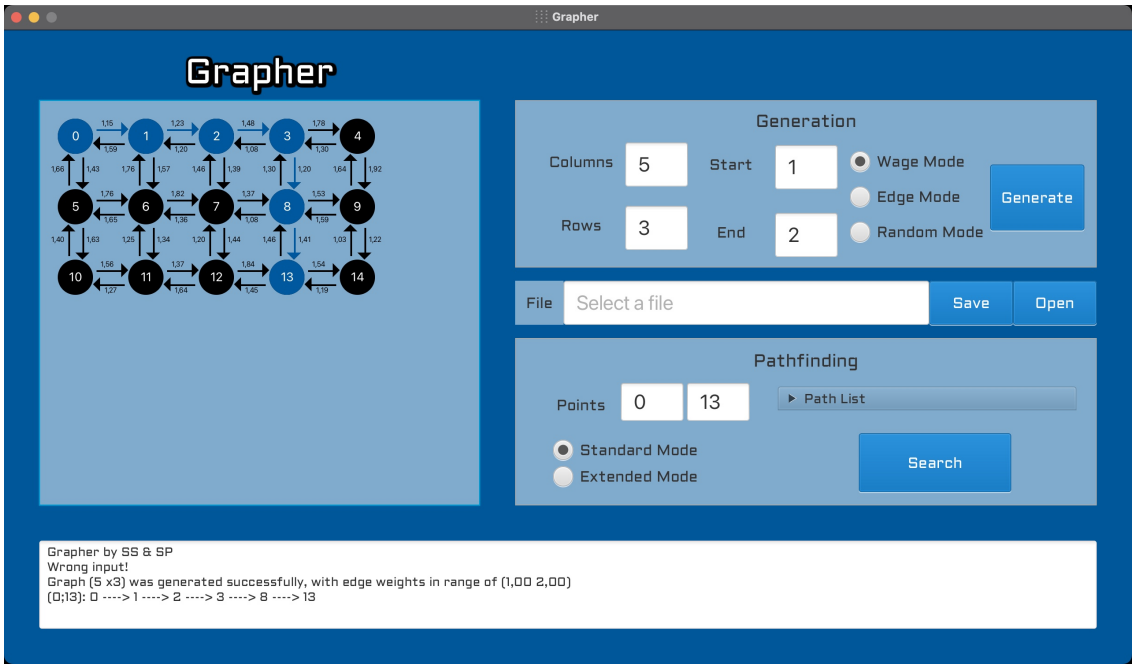
Użytkownik po uruchomieniu aplikacji widzi wyżej pokazany ekran. Ekran ten składa się z pola na rysowanie grafu, okienka z generacją, obsługą pliku oraz szukaniem ścieżek.

## 8.1 Tryb WageMode



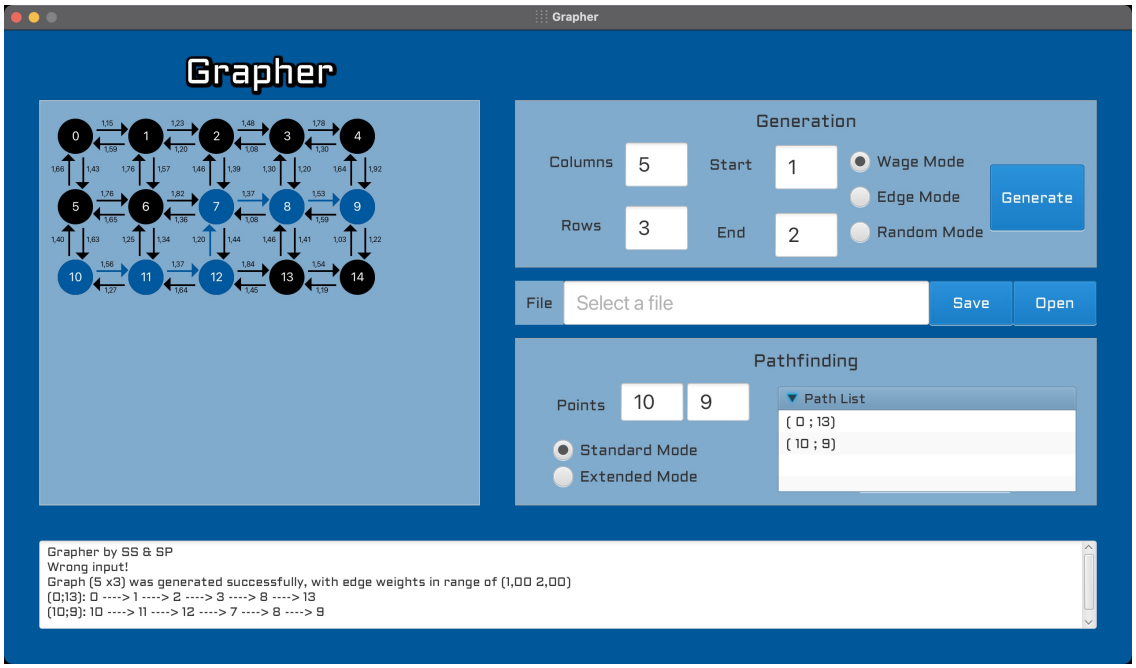

Rysunek 5: Przykładowy graf stworzony w trybie WageMode.

Użytkownik zgodnie z wprowadzonymi danymi może wygenerować graf, jeśli graf posiada poniżej 3600 wierzchołków, graf zostanie wyświetlony po lewej stronie aplikacji.




Rysunek 6: Ścieżka znaleziona w grafie typu WageMode.

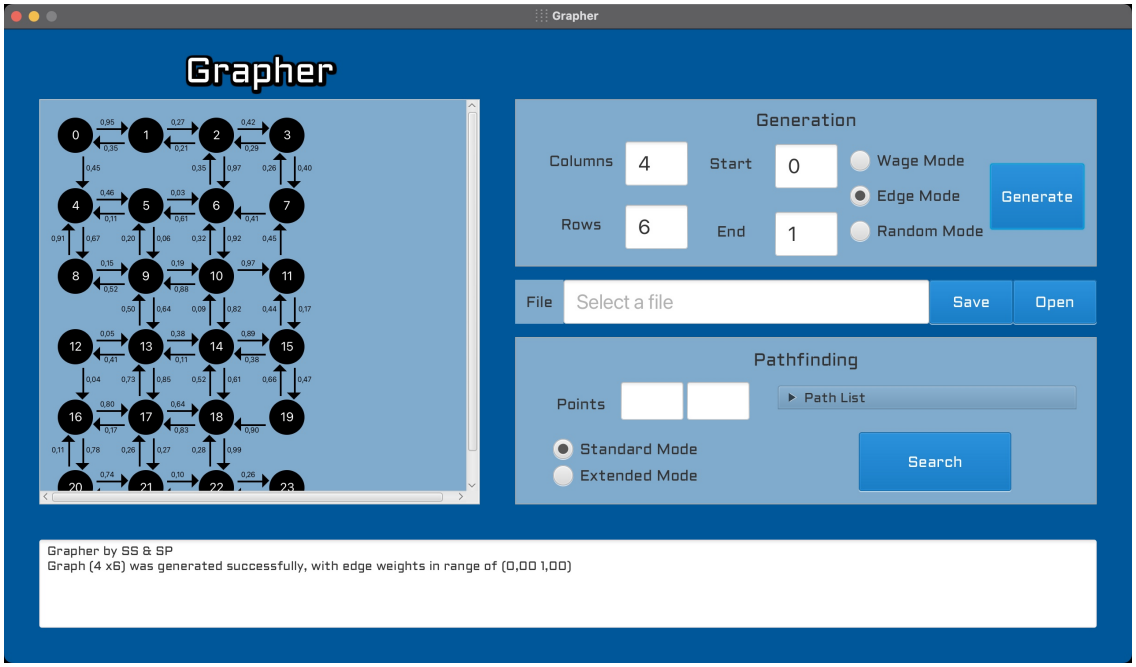
Program umożliwia, też wyszukiwanie ścieżki między zadanymi punktami, co prezentuje powyższy zrzut ekranu.




Rysunek 7: Lista poprzednio znalezionych ścieżek w wygenerowanym grafie.

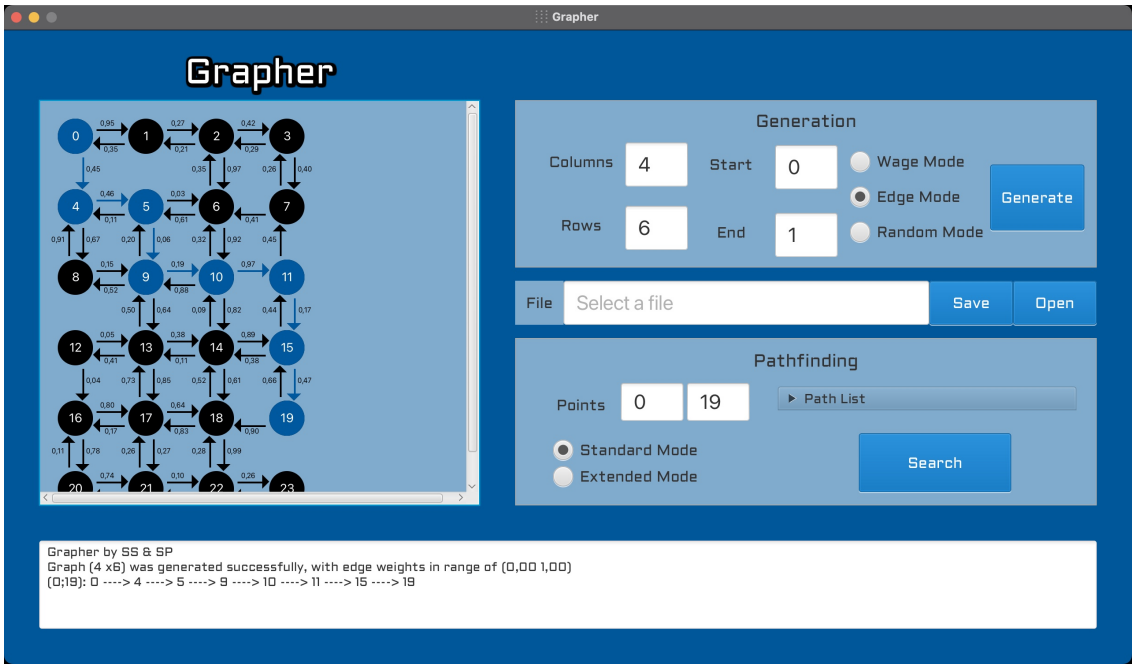
Użytkownik nie traci informacji o znalezionych ścieżkach, wszystkie dotychczasowo znalezione ścieżki na bieżącym grafie zapisane są w pamięci aplikacji i dostępne z poziomu listy ”Path List”

## 8.2 Tryb EdgeMode




Rysunek 8: Przykładowy graf spójny wygenerowany w trybie EdgeMode

Podobnie do generacji zgodnie z trybem WageMode użytkownik ma możliwość generacji i wyświetlania grafu jeśli wygenerowany graf spełnia wymagania wyświetlania.




Rysunek 9: Przykładowe wyszukiwanie najkrótszej ścieżki między punktami w grafie typu Edge-Mode

# 9 Przeprowadzone testy

Program został przetestowany za pomoą testów wykonanych przy pomocy frameworku *JUnit*. Testy można uruchomić za pomocą *Maven'a* albo przy pomocą narzędzi programistycznych wbudowanych w odpowiedni IDE. Przetestowaliśmy wszystkie istotne metody naszego programu i jego wszelkie zachowania.

# 10 Zmiany względem specyfikacji

W niniejszym rozdziale prezentujemy zmiany jakie wprowadziliśmy względem tego co pojawiło się w specyfikacjach.

## 10.1 Klasy i Diagram klas

W trakcie implementacji naszego programu postanowiliśmy zmienić strukturę klas, a co za tym idzie całego diagramu. W rozdziale poświęconemu strukturze programu przedstawiliśmy nowy diagram klas. Tutaj przedstawimy zmiany w klasach.

- EntryData – klasa otrzymała parametr odpowiedzialny za tryb wyświetlania wyświetlania ścieżki oraz dodano nowy konstruktor w tejże klasie,
- Struktura grafu – przechowywanie grafu odbywa się teraz za pomocą klas Graph i Vertex, a nie jak wcześniej Graph, Vertex, Weights, Connections, Existence,
- Nazewnictwo nodes – wszystkie nazwy *nodes* zostały zmienione na *vertex*,
- Klasa GraphIO – owa klasa została rozbita na dwie mniejsze, czyli *GraphReader* i *GraphSaver*,

- Metoda getpath – ta metoda została w pełni usunięta,
- readFromFile – metoda teraz zwraca wczytany graf, a nie przyjmuje tak jak to robiła wcześniej, co poskutkowało zmianą typu z *void* na *Graph*,
- Generacja – klasa ta została usunięte tak samo RandomMode, ponieważ uznaliśmy, że domyślnym trybem jest tryb Random, a Edge i Wage są jego specjalizacją. Postanowiliśmy wykorzystać tutaj interfejs Generator, z którym są zgodne te klasy dziedzicząc po trybie Random implementującym go,
- makeConnectionFromVertex – metoda posiada teraz dostęp do obiekt klasy EntryData, czyli userData.
- Reszta zmian w klasach – wszystkie klasy staraliśmy się jak najbardziej podzielić klasy żeby podzielić odpowiedzialność na mniejsze klasy niż na kilka bardzo dużych.

### 10.2 Obsługa błędów

W naszej specyfikacji implementacyjnej pokazaliśmy tabelę konkretnych błędów obsługiwanych przez nasz program. Ostatecznie postanowiliśmy z nich zrezygnować, ponieważ wyjątki obsługiwane w javie okazały się lepszym sposobem obsługiwania błędów. Wszelkie błędy są obsługiwane za pomocą wyjątków Javy oraz pokazują się odpowiednie komunikaty w graficznym interfejsie użytkownika.

### 10.3 Testowanie programu

W specyfikacji napisaliśmy o wykorzystaniu biblioteki *AssertJ* ale ostatecznie framework *JUnit* zaspokoił w pełni nasze potrzeby związane z testowaniem.

## 11 Podsumowanie współpracy

Współpraca podczas projektu obyła się bez znaczących problemów oraz przebiegała efektywnie. Dzięki ciągłemu kontaktowi mogliśmy dyskutować nasze pomysły na zmiany działania programu oraz jego wyglądu. W kontrolowaniu zmian pomagał nam system kontroli wersji *git*, który pomagał na bezproblemową pracę z repozytorium oraz wprowadzanie w nim zmian.

## 12 Podsumowanie Projektu

Projekt *grapher* był realizowany od 14.04.2022r do 02.06.2022r. W ramach jego powstała dokumentacja projektu, czyli specyfikacja funkcjonalna, implementacja oraz niniejsze sprawozdanie. Oczywiście powstał również sam projekt, który posiada graficzny interfejs uzytkownika. Program umożliwia generowanie grafu w trzech trybach oraz czytanie grafu z możliwością szukania najkrótszej ścieżki między wierzchołkami zadanymi przez użytkownika. Sam sposób wyświetlania ścieżki działa w dwóch trybach jeden pokazuje samą ścieżkę, a drugi pokazuje wagę przejść między wierzchołkami. Program został przetestowany za pomocą testów jednostkowych z wykorzystaniem framework'u JUnit.

## 13 Wnioski

Projekt ten wymagał od nas wiele wysiłku, ponieważ musieliśmy nauczyć się nowych konceptów takich jak przede wszystkim programowanie obiektowe. Wymagał on od nas naszej pierwszej pracy z projektowaniem graficznego interfejsu użytkownika co było dla nas zupełnie nowym i sporym wyzwaniem. To samo tyczy się testowania oprogramowania oraz pracy z programem maven, wymagały one od nas nauki zupełnie nowych technologii i wykorzystanie ich w projekcie wymagało

od nas szybkiej nauki nowych technologii. Z radością możemy stwierdzić, że wszelkie problemy związane z tymi rzeczami stały się dla nas przeszłością i dowodem na to jest w pełni działające oprogramowanie.