# Specyfikacja funkcjonalna programu obsługującego grafy

Anhelina Sudenkova Sebastian Kałuziński

## Cel projektu

Program *graf* ma na celu obsługiwanie grafów. Program działa w trybie nie interaktywnym. Program pozwala na przeszukiwanie grafu przy pomocy algorytmu BFS (*breadth-first search)* jak i również na znalezienie najkrótszej ścieżki przy pomocy algorytmu Dijkstry. Program czyta i zapisuje grafy w ustalonym formacie (rys 1).

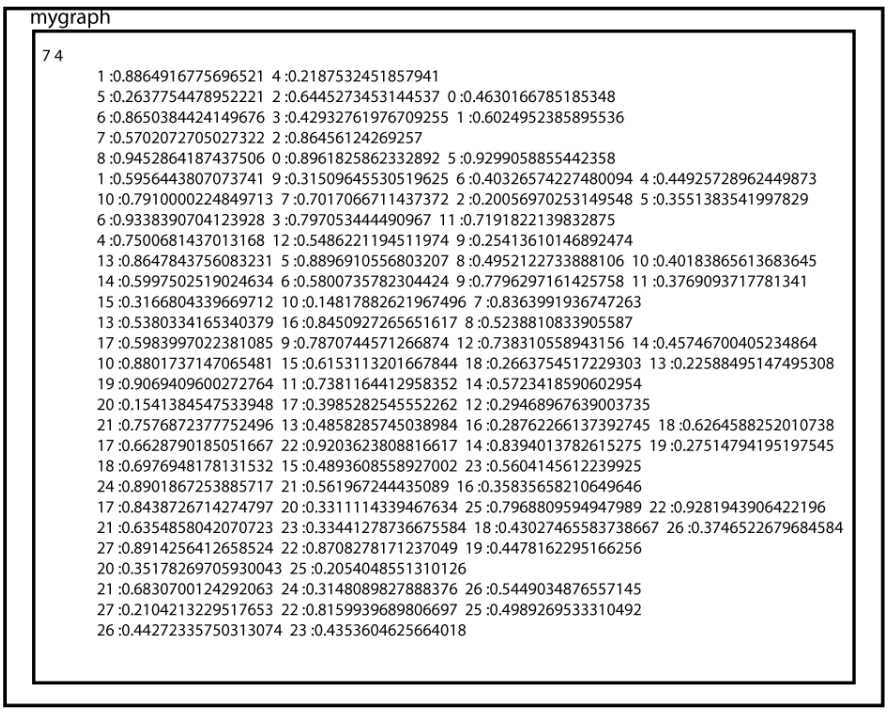
## Dane wejściowe

Jako dane wejściowe program może przyjmować plik, który zawiera informację o grafie (rys 1):

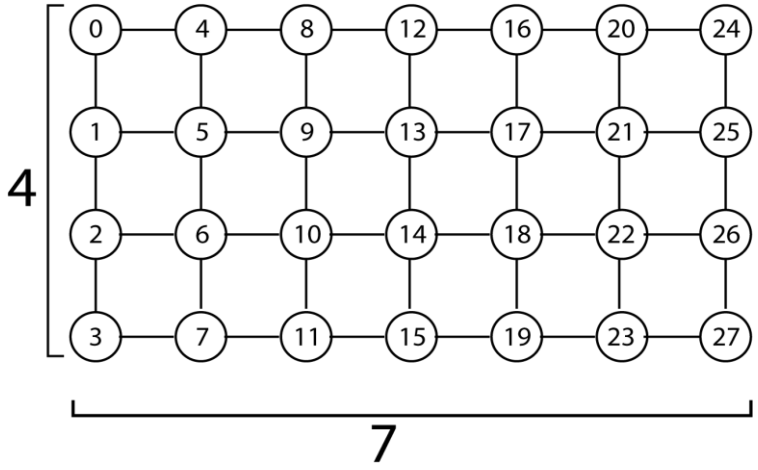
W pierwszej linii dokumentu musi być zapisana ilość kolumn oraz ilość wierszy.

Każda kolejna linia oznacza kolejny wierzchołek grafu (zaczynając od 0) i zawiera informację o sąsiadach tego wierzchołka (z kim jest połączony krawędzią i jaką wagę ta krawędź ma).

Przykład:




Rys.1




Rys.2 (na rysunku nie są pokazane wagi)

Program graf w zależności od danych argumentów skutkuje wypisaniem/utworzeniem odpowiednich komunikatów/plików:

- Dla argumentu „read” nie jest tworzony żaden nowy plik, jedynie na wyjściu dostajemy komunikat zależny od sukcesu działania argumentu.
- Dla argumentu „generate” jest tworzony plik z grafem w razie powodzenia. W przypadku niepowodzenia jest wypisywany odpowiedni komunikat.
- Dla argumentu „checkIntegrity” nie jest tworzony żaden plik, jedynie wypisany komunikat w zależności od sukcesu.
- Dla argumentu „findPath” nie jest tworzony żaden plik, jedynie wypisywany komunikat zależny od sukcesu szukania najkrótszej ścieżki. W przypadku powodzenia wypisane zostają kroki ścieżki oraz „cena” drogi.

**Argumenty wywołania programu**

Program *graf* akceptuje następujące argumenty wywołania:

**--read** *plik* -czyta plik z grafem w ustalonym formacie jednocześnie sprawdzając czy zostały podane odpowiednie wartości połączeń węzłów ( czy znajdują się w odpowiednim zakresie ) ;

--**generate** *min max k w name* generuje graf wielkości **k** kolumn i **w** wierszy mający krawędzie w przedziale ( **min**; **max** )oraz zapisuje go do pliku o nazwie **name**;

**--checkIntegrity** *name* sprawdza czy graf jest spójny ( używając algorytmu przeszukiwaniu grafu wszerz - BFS);

**--findPath** *plik w1 w2* znajduje w danym pliku zawierającym graf najkrótsze ścieżki między w1*(węzeł 1)* a;

*Przykładowe wywołanie programu „graf” :*

- ./graf --generate 5 10 3 3 graf1.txt
  - tworzy graf 3x3 mający krawędzie z przedziału 5-10 oraz zapisuje go pod nazwą „graf1” z rozszerzeniem txt
- ./graf --read grafPrzyklad.txt
  - czyta graf zawarty w pliku „grafPrzykład.txt”
- ./graf --checkIntegrity graf1
  - sprawdza spójność grafu o nazwie graf1
- ./graf –findPath graf1.txt 0 12
  - szuka najkrótszej ścieżki między węzłem 12 a węzłem 14

## Teoria

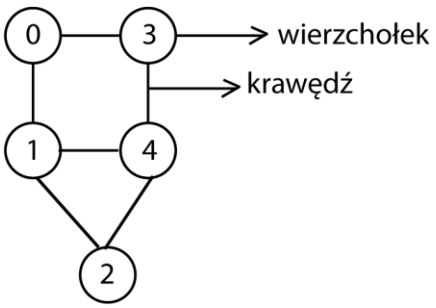
**Graf** (ang. graph) jest strukturą danych, składającą się z dwóch zbiorów:

- zbioru wierzchołków V (ang. vertices),

- zbioru krawędzi E (ang. edges),

matymatycznie zapisujemy w postaci uporządkowanej pary:

G = (V,E).




Rys.3

Z krawędziami grafu mogą być związane dodatkowe wartości, które nazywamy wagami (ang. weight). A graf posiadający takie krawędzi nazywamy **grafem ważonym.**

## Sposoby reprezentacji grafów:

-macierz sąsiedztwa

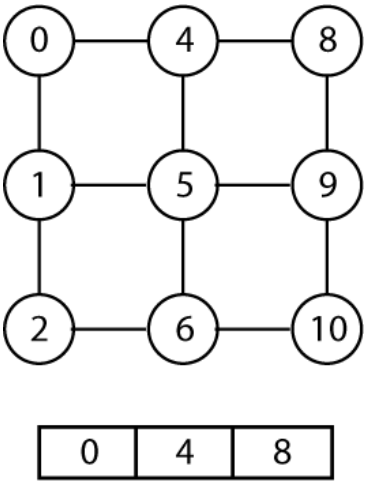
- lista sąsiedztwa

## Przeszukiwanie grafów:

**BFS** (breadth-first search, przeszukiwanie wszerz) polega na znalezieniu listy wierzchołków osiągalnych z wybranego wierzchołka. BFS znajduje drzewo węzłów osiągalnych z podanego wierzchołka i oblicza najkrótsze ścieżki

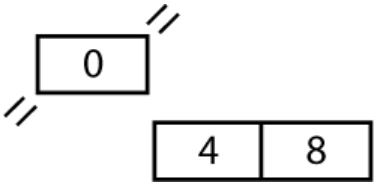
Przykładowe działanie algorytmu: (przyk. 0-5)

1. Tworzymy kolejkę




Rys.4

2. Bierzemy z kolejki kolejny element

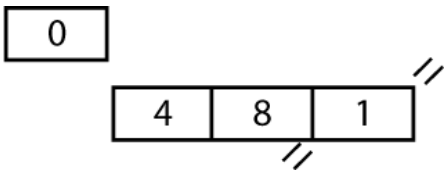



Rys.5

3. Sprawdzamy ten element (czy jest ten element naszym celem?)

TAK NIE

4.A kończymy algorytm4.B dodać sąsiadów tego wierzchołka do kolejki

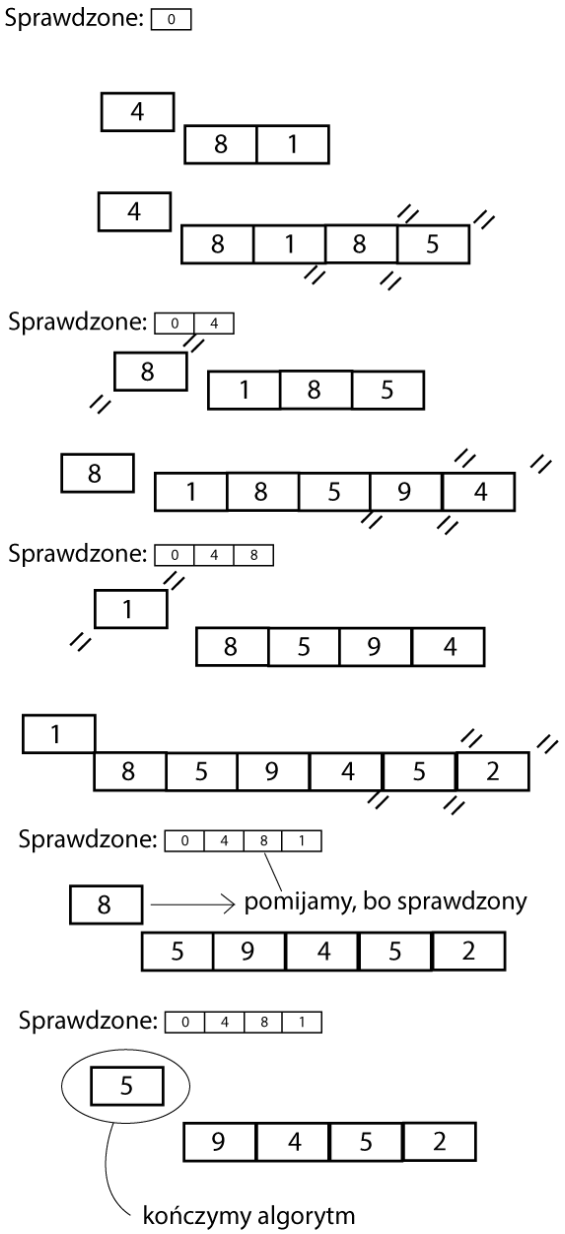



Rys.6

5. Cykl (p.1 -4)

6. Jeżeli kolejka jest pusta, to nie ma szukanego elementu

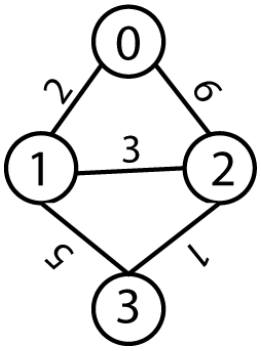
**(UWAGA** trzeba sprawdzać element na wcześniejsze sprawdzanie).




Rys.7

## Algorytm Dijkstry

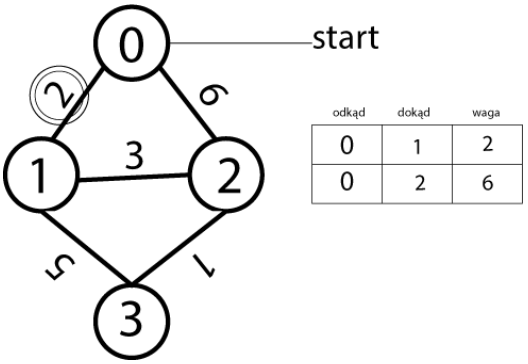
Algorytm Dijkstry służy do wyznaczania najmniejszej odległości od ustalonego wierzchołka s do wszystkich pozostałych w skierowanym grafie. Graf wejściowy nie może zawierać krawędzi o ujemnych wagach.




Rys.8

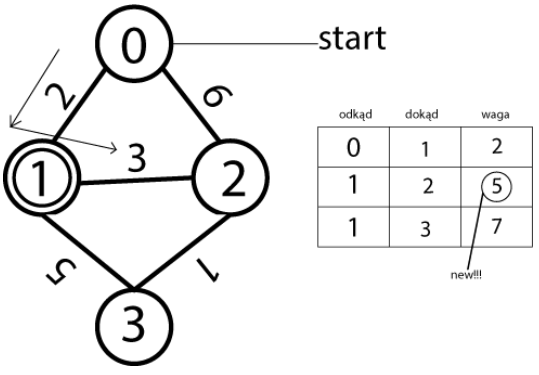
Przykładowe działanie algorytmu:

1.Szukamy krawędź z najmniejszą wagą




Rys.9

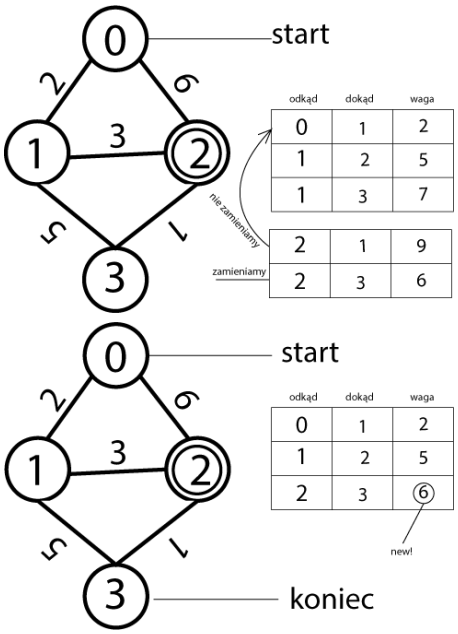
2. Sprawdzamy, czy istnieje mniejsza droga do sąsiadów tego węzła. Jeżeli tak, przypisujemy nową wartość.




Rys.10

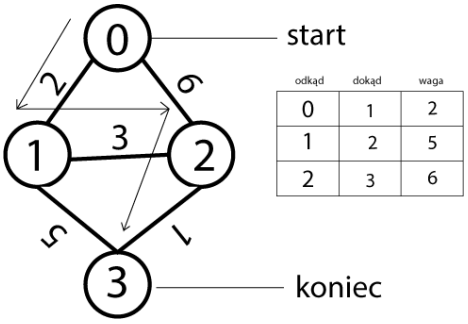
3.Powtarzamy, dopóki to nie będzie zrobione dla wszystkich węzłów grafu.

następny węzel - 2.




Rys.11

4. Wyliczyć końcową drogę.




końcowa droga: 0-1-2-3. Wartość: 6
droga: 0-2-3. Wartość: 7
droga: 0-1-3. Wartość: 7

Rys.12

## Sprawdzanie poprawności danych oraz przebiegu działania:

Argument „read” sprawdza poprawność danych zawartym w pliku podanym razem z argumentem. Czy np. są one dodatnie. Jeżeli nie to podaje między, którymi węzłami została podana zła wartość oraz w, którym wierszu pliku z danymi znajduje się błędna wartość. Zostają wypisane komunikaty o wszystkich błędnych wartościach po czym program kończy działanie z błędem. W przypadku powodzenia wypisujemy komunikat „Przeczytano graf zapisany w pliku nazwa.txt z powodzeniem. Ma on k kolumn oraz w wierszy.”

Argument „generate” tworzy graf o wielkościach k(kolumny) oraz w(wiersze), które są podane po argumencie „generate”. Przy tworzeniu zostaje sprawdzone czy mamy odpowiednią ilość miejsca aby przechować graf o danej wielkości. Jeśli nie to wiemy, że nastąpił przypadek OOM( Out Of Memory ). Wypisujemy odpowiedni komunikat i kończymy działanie programu. W przypadku pomyślnego utworzenia wypisujemy komunikat „Utworzono graf o k kolumnach oraz w wierszach. Został on zapisany pod podaną nazwą : nazwa.txt .”

Argument „checkIntegrity” sprawdza spójność grafu wykorzystując algorytm przeszukiwania wszerz (BFS). Wykorzystywane jest tez działanie modułu „read”. Dane są przypadki:

- Podaliśmy przyjmowany przez moduł „read” graf –
    a) Graf jest spójny (istnieje połączenie miedzy każdymi dwoma węzłami). W takim przypadku zostaje wypisany komunikat o spójności grafu.
    b) Graf nie jest spójny (nie istnieje połączenie miedzy każdymi dwoma węzłami). W takim przypadku zostaje wypisany komunikat o braku spójności grafu.
- Podaliśmy nie istniejący graf lub nie jest on przyjmowany przez moduł „read”– Zostaje zwrócony odpowiedni komunikat błędu.

Argument „findPath” znajduje najkrótszą ścieżkę między dwoma wierzchołkami, które są podane po argumencie (przykład poprawnego wywołania w dziale **Argumenty** wywołania programu). Argument ten zachowuje się w podobny sposób do „checkIntegrity”. Używa działania argumentów „read”. Dane są przypadki:

- Podaliśmy przyjmowany przez moduł „read” graf –
    a) Istnieje najkrótsza ścieżka między podanymi węzłami, zostaje zwrócony jej koszt oraz przebieg.
    b) Nie istnieje ścieżka między podanymi węzłami, zostaje zwrócony odpowiedni komunikat i błąd.
- Podaliśmy nie istniejący graf lub nie jest on przyjmowany przez moduł „read”– Zostaje zwrócony odpowiedni komunikat błędu.
- Gdy podano wierzchołek, który nie znajduje się w grafie(np. podano wierzchołek 26 gdy mamy wygenerowany graf 5x5). W takich przypadku zostaje wypisany odpowiedni komunikat z wielkością grafu oraz , który wierzchołek jest niepoprawny.

# Specyfikacja implementacyjna

## Informacje ogólne

Możliwe są dane parametry uruchomienia:

**--read <**plik>
Odpowiada za sprawdzenie poprawności grafu oraz wypisanie go w konsoli poleceń

**--generate** <k> <w> <min> <max> <name>
Odpowiada za tworzenie grafu, wymagana jest ilość kolumn, wierszy oraz nazwa pod jaką należy zapisać plik.

**--checkIntegrity** <name>
Odpowiada za sprawdzenie spójność (każdymi dwoma węzłami) grafu zapisanego w podanym pliku.

**--findPath** <name> <w1> <w2>
Odpowiada za znalezienie najkrótszej ścieżki w grafu pod zapisanego pod nazwą name.txt.

## Używane biblioteki

**stdio.h -** obsługa strumieni wejściowych i wyjściowych programu.
**stdlib.h** – obsługa funkcji rand, free, malloc, realloc, srand.
**string.h** – rozszerzona obsługa ciągów znaków (nazwy plików).
**time.h** – pozwala na tworzenie pseudolosowych liczby do grafów.

## Opis modułów

(Rys 12)

### main.c

Importuje prototypy funkcji zawarte w ***graph.h***, które są zgodne z ich definicjami. Wykorzystuje ich działanie w odpowiednich przypadkach.

### bfs.h

Tworzy nowy typ zmiennych ***queueNode*** oraz ***Queue,***.
Zawiera prototypy wszystkich funkcji używanych w pliku bfs.c oraz funkcji obsługujących kolejkę.

### dijkstra.h

Tworzy nowy typ zmiennych ***pNode*** będącej elementem kolejki priorytetowej.
Zawiera prototypy wszystkich funkcji używanych w dijkstra.c oraz funkcji obsługujących kolejkę priorytetową wraz z funkcjami związanymi z algorytmem dijkstry.

### graph.h

Tworzy nowy typ zmiennych ***Graph***.
Zawiera prototypy wszystkich funkcji używanych w naszej strukturze plików.

**bfs.c**
Zawiera definicję funkcji ***bfs(int start, Graph graph)***, która przyjmuje zmienną, której typ jest opisany w **graph.h** typu *Graph* o nazwie *graph* numer węzła startowego. Funkcja sprawdza spójność grafu zapisanego w parametrze *graph* wykorzystując algorytm BFS.

**dijkstra.c**
Zawiera definicję funkcji ***findPath(Graph graph, int start,int end)***, która przyjmuje zmienną, której typ jest opisany w **graph.h** typu *Graph* o nazwie *graph* numer węzła startowego i końcowego oraz funkcje obsługującą implementację kolejki priorytetowej. Funkcja ***findPath*** szuka wybiera najkrótszą ścieżkę zawierającą start oraz koniec, jeśli takowa nie istnieje to zwraca odpowiedni komunikat.

**generate.c**

Zawiera definicję funkcji ***generateGraph(double min,double max,int columns, int rows)*** oraz ***readGraph(char* nameOfFile) ,generateGraph(double min, double max, int columns, int rows)*** przyjmuje odpowiednio wartość minimalną krawędzi, wartość maksymalna krawędzi, ilość kolumn oraz rzędów grafu, który ma zostać wygenerowany. Funkcja po wygenerowaniu zwraca zmienną typu ***Graph*** opisany w ***graph.h.***

***readGraph(char* nameOfFile)*** przyjmuje nazwę pliku, który ma zostać odczytany, zostaje wypisany komunikat oraz zwrócona zmienna typu ***Graph*** w, której są przeczytywane przeczytane wartości z pliku.

## Testowanie

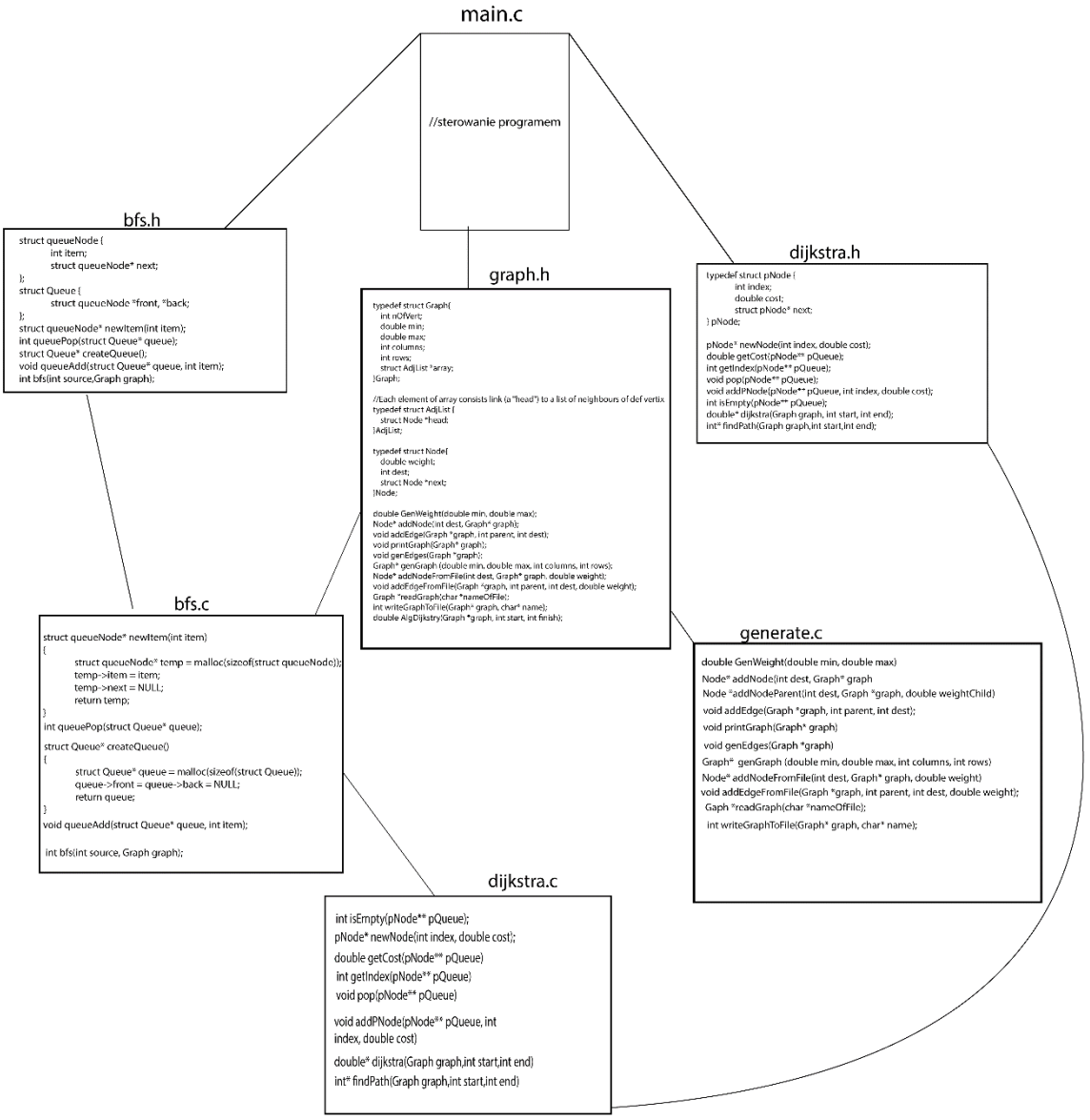
Aby przetestować program zostaną sprawdzone dane warunki brzegowe (z ang. Edge cases) przy użyciu pliku reguł make przy odpowiednich warunkach i odpowiednich danych:

- Czytanie grafu
  - Plik ma odpowiedni format oraz krawędzie mają odpowiednie wartości.
  - Wartość krawędzi jest niedodatnia.
  - Plik jest pusty.
- Generowanie grafu
  - Wystąpił przypadek OOM w wyniku braku miejsca lub problemu z alokacją miejsca dla zmiennych przy użyciu *malloc*.
  - Poprawna generacja grafu.
- Sprawdzanie spójności grafu
  - Graf nie jest spójny.
  - Graf jest spójny.
  - Plik zawierający graf nie istnieje.
- Szukanie najkrótszej ścieżki w grafie
  - Znaleziono poprawnie najkrótszą ścieżkę w grafie.
  - Plik zawierający graf nie istnieje.
  - Podane węzły nie są zawarte w grafie.
  - Nie ma połączenia do danego węzła w grafie

## Konwencja

W czasie pisania programu zostanie wykorzystana konwencja *camelCase* z wyjątkiem nazwy dla nowo definiowanej zmiennej ***Graph*** z racji na jej naturę

# Diagram zależności między plikami




Rys. 12