Specyfikacja implementacyjna
programu grapher

Szymon Półtorak i Sebastian Sikorski

17.03.2022r

Streszczenie

W dokumencie zawarte są informacje na temat implementacji projektu dotyczącego *grafów*, napisanego w języku *C*. Przedstawiane są metody implementacji, podział na moduły, środowisko w jakim program został napisany oraz wyjaśnione jest jego działanie.

Spis treści

1 Cel projektu - streszczenie

Program ma za zadanie generować pliki z grafami o ustalonym formacie oraz czytanie takich plików w celu znalezienia najkrótszej ścieżki między podanymi przez użytkownika punktami. Grapher posiada cztery tryby:

- Wage Mode – program generuje graf spójny o losowych wagach krawędzi,
- Edge Mode – program losuje istnienie krawędzi między wierzchołkami grafu oraz wagi do momentu powstania grafu spójnego,
- Random Mode – program losuje istnienie krawędzi i ich wagi. W tym trybie graf może być niespójny,
- Read Mode – program czyta plik o ustalonym formacie i szuka drogi pomiędzy podanymi przez użytkownika punktami.

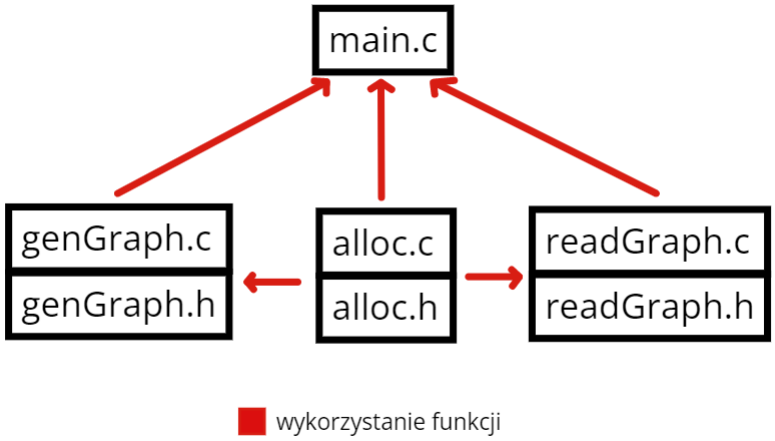
Po szczegółowe wyjaśnienie funkcjonalności trybów, formatu pliku z grafem, struktury folderu oraz składni programu odsyłamy do specyfikacji funkcjonalnej projektu *grapher*.

2 Środowisko pracy

Język programowania	C17 (ISO/IEC 9899:2018)
Kompilator	gcc 10.2.1
System Operacyjny	Windows Subsystem for Linux, Debian 11
Środowisko programistyczne	Visual Studio Code 1.65.2
System kontroli wersji	Git 2.30.2

3 Diagram modułów

Program składa się z trzech modułów: `genGraph`, `alloc` i `readGraph`. Składają się one z dwóch plików każdy. Jednym z nich jest plik źródłowy *.c*, a drugi to plik nagłówkowy *.h*. Oprócz nich znajduje się również plik `main.c`, zawierający główną funkcję sterującą programem.



Rysunek 1: Diagram modułów.

4 Przedstawienie poszczególnych modułów

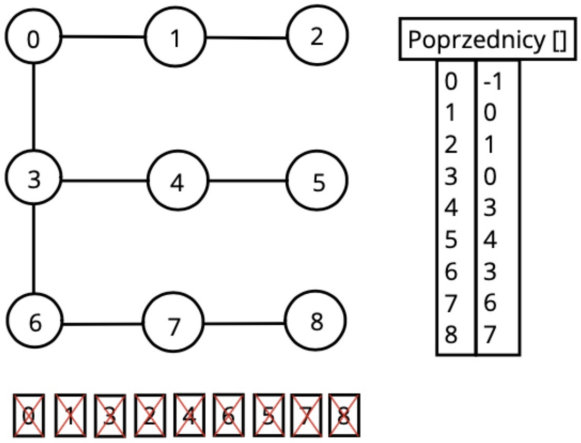
Plik `main.c` odpowiada za sterowanie całym programem i wywołuje potrzebne funkcje z odpowiednich modułów. Moduły programu i ich rola:

- **Moduł alloc** – znajdują się w nim funkcje pomogające programowi w alokowaniu i uwalnianiu pamięci z tablic i struktur,
- **Moduł genGraph** – zawiera on funkcje odpowiedzialne za generowanie grafu w trzech trybach oraz funkcje pomagające wykonanie odpowiednich procedur w danym trybie,
- **Moduł readGraph** – posiada on funkcje, których zadaniem jest wczytywanie grafu, sprawdzanie jego spójności oraz znajdowanie w nim najkrótszej ścieżki między zadanymi punktami.

5 Wykorzystane algorytmy

5.1 Breadth-first search(BFS)

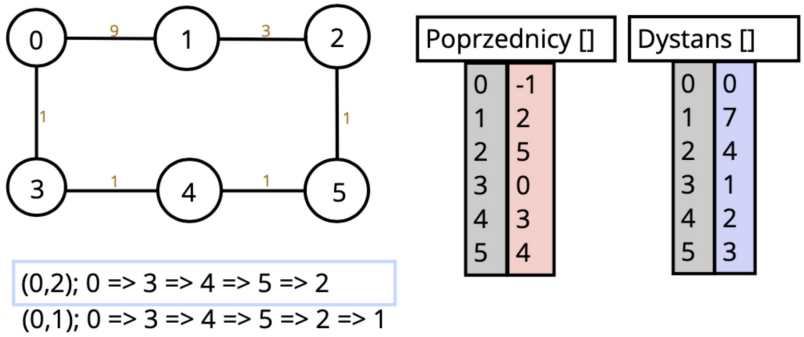
Sprawdzanie spójności grafu rozpoczyna się z dowolnego wierzchołka, w przypadku naszego algorytmu domyślnym wierzchołkiem jest wierzchołek zero. Algorytm w celu sprawdzenia spójności tworzy tablicę poprzedników o długości odpowiadającej ilości wierzchołków oraz zapełnia ją wartościami -1. Rozpoczynając 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: Przykładowy graf wraz z tablicą poprzedników.

5.2 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 trzech tablic przechowujących informacje odpowiednio o tym czy wierzchołek odwiedzono, dystansie jakiego dotąd wymagało odwiedzenie oraz poprzedniku tego odwiedzenia. W przeciwieństwie do algorytmu BFS, algorytm ten korzysta z kolejki priorytetowej, gdzie wartością decydującą jest dotychczasowa odległość do punktu. Dalej analogicznie następuje iterowanie po wszystkich wierzchołkach aż do momentu, w którym kolejka jest pusta. 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 3: Przykładowy wynik najkrótszej ścieżki w algorytmie Dijkstry.

6 Wykorzystane struktury

Poniżej prezentujemy najważniejsze struktury wykorzystane przez nasz program:

- `struct entryRead`:

```
struct entryRead {
    char* fileName;
    bool printFlag;
    int* points;
} entryR;
```

- `struct entryGen`:

```
struct entryGen {
    int rows;
    int columns;
    double rangeStart;
    double rangeEnd;
    short int mode;
    char* fileName;
} entryG;
```

- `struct node`:

```
struct node {
    bool* edgeExist;
    double* edgeWeight;
} node;
```

- `struct graphRead`:

```
struct graphRead {
    node** graph;
    int rows;
    int columns;
} graphR;
```

7 Wykorzystane funkcje

W naszym programie zaimplementowane zostało wiele funkcji oraz struktur. Poniżej prezentujemy tylko te najważniejsze z nich:

7.1 Moduł genGraph

- `void generateMode(struct entryGen)` – funkcja główna odpowiadająca za generowanie grafu w wybranym przez użytkownika trybie,
- `bool checkIfCoherentGen(node** graph, entryG* entry)` – zwraca wartość false lub true w zależności, czy graf jest spójny, czy nie,
- `void saveGraphToFile(entryG* entry, node** graph)` – odpowiada za zapisanie gotowego grafu do pliku,
- `double generateWeights(entryG* entry)` – jej zadaniem jest wygenerowanie wagi z podanego przez użytkownika przedziału,

7.2 Moduł alloc

- `entryG* allocEntryGen(char **argv)` – zwraca wskaźnik na strukturę typu `entryG`,
- `entryR* allocEntryRead(int argc, char **argv)` – funkcja zwraca wskaźnik na strukturę typu `entryR`,

7.3 Moduł readGraph

- `void readMode(struct entryRead)` – funkcja główna odpowiadjąca za odczytanie pliku i znalezienie najkrótszej ścieżki między podanymi przez użytkownika punktami,
- `graphR** readFromFile(entryR* entry)` – wczytuje graf z odpowiednio sformatowanego pliku do tablicy,
- `int checkIfCoherentRead(graphR** graph, entryR* entry)` – zwraca wartość 0 lub 1 w zależności, czy graf jest spójny, czy nie,
- `void findPath(graphR** graph, entryR* entry)` – odpowiada za znalezienie najkrótszej ścieżki między podanymi przez użytkownika punktami,
- `void printShortPath(entryR* entry, int* parents)` – wyświetla najkrótszą ścieżkę w podstawowym formacie,

- `void printExtendedPath(entryR* entry, int* parents, double* weights)` – wyświetla najkrótszą ścieżkę w rozszerzonym formacie,

8 Przeprowadzanie zmian oraz testów

Do pracy z projektem wykorzystujemy system kontroli wersji *git*. Wszelkie zmiany w programie będą przekazywane do platformy Projektor za pośrednictwem systemu kontroli wersji w postaci commitów. Testowanie programu będzie przeprowadzone w sposób półautomatyczny. Do tego będziemy wykorzystywać odpowiednią formułę z pliku *makefile*. Formuła to:

- **make test** – formuła odpowiedzialna za skompliowanie plików z folderu test oraz plików bedących modułami w programie grapher. Dokonuje ona komplacji do pliku wykonywalnego `./test`, a następnie go uruchamia.