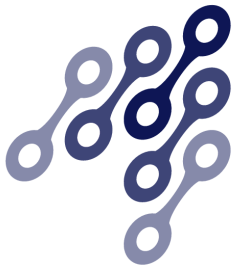
POLITECHNIKA CZĘSTOCHOWSKA

WYDZIAŁ INŻYNIERII MECHANICZNEJ I INFORMATYKI



PRACA MAGISTERSKA

Implementacja podstawowej biblioteki grafów w języku C

Implementation of the basic graph library in C

Patryk Kwiatkowski

Nr albumu: 101510

Kierunek: Informatyka

Studia: stacjonarne

Stopień: II

Promotor pracy: dr inż. Ireneusz Szcześniak

Praca przyjęta dnia:

Podpis promotora:

Częstochowa, 2012

Serdeczne podziękowania
dla Pana dra Ireneusza Szcześniaka
za poświęcony mi czas oraz nieocenioną pomoc
w przygotowaniu niniejszej pracy dyplomowej.

Spis treści

Cel pracy

Celem pracy jest stworzenie oprogramowania komputerowego umożliwiającego tworzenie aplikacji, wykorzystujących w swym działaniu elementy matematycznej teorii grafów. Oprogramowanie to, dalej zwane *biblioteką* powinno dostarczyć interfejs programistyczny (API), pozwalający na tworzenie oraz działanie na podstawowych elementach grafów. Biblioteka ta powinna również implementować jeden z wybranych algorytmów teorii grafów.

Oprogramowanie to powinno poprawnie działać na systemach operacyjnych rodziny Linux, przy zachowaniu jak najmniejszych wymagań na ich zasoby pamięciowe oraz obliczeniowe. Rozwiązanie to powinno również dostarczać mechanizm testów jednostkowych — bazujących na dowolnej platformie testowej — pozwalających na dalszy rozwój oprogramowania.

Spełniając powyższe wymagania, biblioteka ta powinna zostać stworzona przy użyciu języka C. Zachowując przy tym prostotę oraz przejrzystość kodu źródłowego. Kod ten powinien być również dobrze udokumentowany, zarówno za pomocą poniższej pracy jak i wygenerowanej automatycznie dokumentacji.

Wstęp

Systemy nawigacji GPS, sieć Internet, gry komputerowe, translatory języków obcych, biologia, chemia, socjologia — to wszystko, i wiele innych dziedzin życia, łączy jeden wspólny element — teoria grafów. Kiedy rozwój informatyki pozwolił na reprezentowanie grafów za pomocą komputera, okazało się, że algorytmy na nich oparte znajdują wiele praktycznych zastosowań. Grafy są jednymi z najbardziej wszechobecnych modeli zarówno świata naturalnego jak i stworzonego przez człowieka.

Oprogramowanie oparte na analizie grafów znalazło zastosowanie w wyznaczaniu trasy pomiędzy punktami na mapie, czy najszybszej drogi ewakuacji z kompleksu budynków. Przedstawienie sieci komputerowych w postaci grafów pozwoliło na stworzenie programów usprawniających przepływ pakietów w Internecie. Ta dziedzina matematyki jest równie przydatna w biologii, gdzie wierzchołek może reprezentować regiony, w których niektóre gatunki istnieją, a krawędzie ścieżki migracji. Informacja ta jest ważna, gdyż patrząc na powstałe wzorce, można zbadać wpływ rozprzestrzeniających się chorób, pasożytów czy zmiany ruchów na inne zwierzęta.

Dzięki możliwościom jakie dają dzisiejsze komputery w przetwarzaniu informacji, powstało wiele bibliotek obsługujących obliczenia oparte o teorię grafów. Rozwiązania te pojawiają się w niemal każdym języku programowania, od *C++*, przez *D*, aż do *Pythona* czy *Matlaba*. Tematyka teorii grafów, ze względu na szeroką gamę zastosowań oraz dużą przydatność — zwłaszcza przy analizie, projektowaniu oraz udoskonalaniu sieci komputerowych — została wybrana przez autora, jako temat przewodni niniejszej pracy dyplomowej. Mimo istnienia dużej ilości bibliotek komputerowych implementujących zagadnienia tej tematyki, niewiele z nich cechuje się prostotą oraz przejrzystością kodu źródłowego. Zaś wiele z nich zazwyczaj zużywa wiele zasobów pamięciowych, oraz działa względnie powolnie. Potrzeba ograniczenia zużywanych zasobów, oraz stworzenia projektu prostego i czytelnego, była główną motywacją autora do podjęcia się stworzenia „*Implementacji podstawowej biblioteki grafów w języku C*”.

Rozdział pierwszy niniejszej pracy zawiera wiedzę teoretyczną posiadaną, lub zebraną, przez autora pracy na potrzeby realizacji wcześniej przedstawionych celów. W rozdziale tym wymienia się i opisuje pojęcia związane z szeroko pojętą teorią grafów, kładąc szczególny nacisk na wykorzystywane później jej elementy.

Drugi rozdział pracy zawiera szczegółowy opis zaprojektowanej biblioteki. Wyjaśnia powody wyboru konkretnych technologii do realizacji obranych celów. W rozdziale tym przedstawiona została budowa poszczególnych modułów, sposób przechowywania informacji oraz metody usprawniające pracę przy dalszym rozwoju biblioteki. Zawiera on też kilka przykładów wykorzystania interfejsu użytkownika.

Ostatnia część pracy przedstawia porównanie stworzonego na cele pracy projektu, z istniejącymi już rozwiązaniami. Porównania te zostały przeprowadzone pod kątem ilości zużywanej pamięci, oraz czasu jaki jest potrzebny na wykonanie podstawowych funkcji tego typu biblioteki.

Dodatkowo na końcu pracy, jako załącznik, zamieszczone zostały przykłady użycia stworzonego oprogramowania, wraz z krótkimi komentarzami w języku angielskim.

1. Wybrane zagadnienia teorii grafów

Rozdział ten, powstał w celu zdefiniowania oraz wyjaśnienia elementarnych pojęć związanych z teorią grafów. Wiadomości w nim zawarte opierają się na publikacjach [1] oraz [2], w których można odnaleźć więcej szczegółowych informacji. Pojęcia te będą często używane w dalszych częściach pracy, zatem usystematyzowanie tej wiedzy jest niezwykle istotne, aby dobrze zrozumieć sens przekazywanych słów.

Teoria grafów dział matematyki i informatyki zajmujący się badaniem własności grafów, matematycznych struktur wykorzystywanych do modelowania relacji pomiędzy obiektami.

Graf (ang. *graph*) *G*, struktura matematyczna składająca się z niepustego zbioru skończonego *V(G)*, którego elementy nazywamy wierzchołkami i skończonego zbioru *E(G)* różnych par różnych elementów zbioru *V(G)*, które nazywamy krawędziami.

Wierzchołek (ang. *vertex*), inaczej węzeł, element składowy grafu, reprezentuje obiekt rzeczywisty, punkt odniesienia, dzięki czemu krawędzie między węzłami mogą reprezentować relacje. Często numerowany, może jednak posiadać etykietę (nazwę).

Krawędź (ang. *edge*) łączy ze sobą dwa wierzchołki grafu (w szczególnym wypadku wierzchołek sam ze sobą). Może posiadać kierunek (krawędź skierowana), lub nie (krawędź nieskierowana). Często posiada wagę/koszt, czyli przypisaną liczbę, która oznaczać może odległość między węzłami.

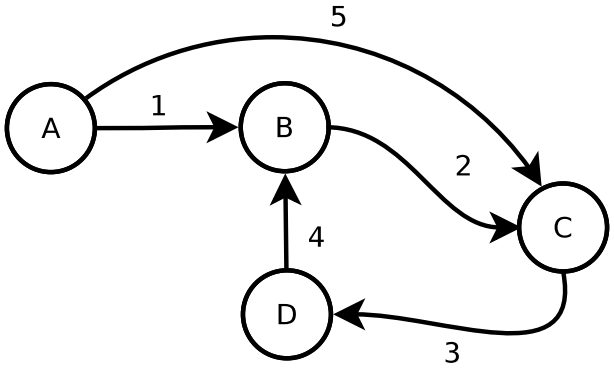
Droga inaczej ścieżka to trasa wyznaczana przez krawędzie, polegająca na podróżowaniu od wierzchołka do wierzchołka po łączących je krawędziach.

Sąsiad dwa wierzchołki grafu, pomiędzy którymi istnieje krawędź.

Ze względu na kierunkowość krawędzi, możemy wyróżniamy trzy podstawowe rodzaje grafów:

- Nieskierowane (grafy proste) — wszystkie krawędzie grafu są nieskierowane.
- Skierowane (digrafy) — wszystkie krawędzie grafu są skierowane.
- Mieszane — może zawierać jednocześnie krawędzie skierowane i nieskierowane.

Przedstawiając obrazowo powyższe definicje można posłużyć się prostym rysunkiem:



Rysunek 1. Przykład prostego grafu skierowanego

Punkty A, B, C oraz D znajdujące się na rysunku 1 nazywamy *wierzchołkami*, łączące je linie *krawędziami* (skierowanymi), zaś całość tworzy strukturę *grafu* (skierowanego). Krawędzie o wagach 1, 2 oraz 3 tworzą *drogę*, z węzła A do węzła D. Rozpoczynając trasę z wierzchołka D nie mamy możliwości przedostania się do węzła A. *Sąsiadem* wierzchołka B jest wierzchołek C.

Wszystkie grafy mogą być reprezentowane na wiele sposobów. Najbardziej naturalnym i najprostszym dla człowieka jest *rysunek* grafu, jednakże jest to forma reprezentacji, której komputery nie potrafią (jeszcze) zrozumieć. Innymi metodami zapisu mogą być:

- Macierz sąsiedztwa — macierz kwadratowa o rozmiarze n, równym ilości wierzchołków w grafie. Każdy jej element oznacza liczbę krawędzi łączącą i-ty i j-ty węzeł. Tak zaimplementowana komputerowa struktura danych gwarantuje, że operacje sprawdzenia, czy dodania oraz usunięcia krawędzi odbywają się w stałym czasie. Do jej wad należy duża ilość potrzebnej pamięci – $O(n^2)$.
- Lista sąsiedztwa — dla każdego wierzchołka zapamiętywana jest lista sąsiadujących z nim węzłów. Metoda ta wymaga ilości pamięci proporcjonalnej do liczby krawędzi, także czas potrzebny na przejrzenie całego zbioru krawędzi jest proporcjonalny

do jego rozmiaru. Wadą jest tu zwiększona złożoność operacji elementarnych (np. usunięcie krawędzi).

- Macierz incydencji — macierz o wymiarach odpowiadających ilości węzłów na ilość krawędzi. Zawiera jedynie informacje takie, że wartość w punkcje {i, j} = 1 tylko, gdy j-ta krawędź zaczyna się na i-tym wierzchołku, = -1 gdy się kończy, a 0 gdy nie są incydentne.

Dla lepszego zrozumienia owych struktur warto ponownie przeanalizować rysunek 1 oraz odpowiadające mu metody zapisu:

- macierz sąsiedztwa: $\begin{vmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{vmatrix}$
- lista sąsiedztwa: $\begin{aligned} &A \to B,\ C \\ &B \to C \\ &C \to D \\ &D \to B \end{aligned}$
- macierz incydencji: $\begin{vmatrix} 1 & 0 & 0 & 0 & 1 \\ -1 & 1 & 0 & -1 & 0 \\ 0 & -1 & 1 & 0 & -1 \\ 0 & 0 & -1 & 1 & 0 \end{vmatrix}$

Informacje przytoczone w powyższym rodziale, są podstawowymi pojęciami teorii grafów, stanowią one jedynie ułamek tej rozległej dziedziny nauki, lecz pomogą zrozumieć tematykę poruszaną przez niniejszą pracę.

1.1. Algorytm Dijkstry

Problemem najkrótszej ścieżki, określamy chęć odnalezienia takiego połączenia między wierzchołkami, które ma najmniejszy koszt przejścia, przy czym koszt przejścia określony

jest jako suma wag na krawędziach ścieżki[3]. Problemy te rozwiązuje się w oparciu o grafy spójne[1] oraz ważone[2].

Problem ten możemy zobrazować jako próbę znalezienia najszybciej drogi dla pakietów pomiędzy dwoma routerami w sieci. Każde z urządzeń trasujących reprezentowane jest wówczas przez węzeł grafu a waga krawędzi może oznaczać przepustowość łącza pomiędzy nimi (informacja przydatna dla protokołów trasowania stanu łącza, np. *OSPF*[4]).

Algorytm opracowany przez holenderskiego informatyka Edsgera Dijkstrę, służy do rozwiązywania tego typu problemów. Jest jednym z najważniejszych algorytmów teorii grafów, wykorzystywanym m. in. właśnie w protokole trasowania *OSPF*. Warunkiem ograniczającym jego działanie jest wymóg istnienia wyłącznie nieujemnych wag krawędzi[3]. Mając dany graf z wyróżnionym wierzchołkiem (źródłem) algorytm znajduje odległości od niego do wszystkich pozostałych węzłów.

Listing 1.1. Pseudokod algorytmu Dijkstry[3]

```
1  for each vertex v in V[G]
2    d[v] := infinity
3    p[v] := undefined
4  end for
5  d[s] := 0
6  Q := all vertices array
7
8  while (Q is not an empty)
9    u := Extract-Min(Q)
10   for each edge (u,v) outgoing from u
11     if (d[v] > d[u] + w(u,v))
12       d[v] := d[u] + w(u,v)
13       previous[v] := u
14     end if
15 end while
```

W algorytmie tym pamiętany jest zbiór `Q` wierzchołków, dla których nie obliczono jeszcze najkrótszych ścieżek, wektor `d` odległości od wierzchołka `s` (źródłowego) do i-tego, oraz wektor poprzedników `p`, dzięki któremu można odtworzyć odnalezioną ścieżkę. Początkowo (wiersze: 1 — 6) zbiór `Q` zawiera wszystkie wierzchołki, wektor `d` jest wy-

[1]Graf możemy nazwać spójnym wówczas, gdy dla każdej pary wierzchołków istnieje droga pomiędzy nimi.

[2]Graf możemy nazwać ważonym wówczas, gdy każda z jego krawędzi posiada atrybut wagi/kosztu.

[3]Algorytm Forda-Bellmana pozbawiony jest tej wady, jednakże przez to charakteryzuje się dużo większą złożonością czasową[1]

pełniany wartościami nieskończonymi a `p` niezdefiniowanymi (np. `NULL`) Odległość dla wierzchołka źródłowego wynosi zawsze zero. Algorytm analizuje zawsze węzły o najmniejszej wartości `d[v]`, czyli te najbliższe wierzchołkowi źródłowemu. Zapewnia to operacja `Extract-Min(Q)` w wierszu 9 listingu 1.1, która dodatkowo usuwa pobrany element ze zbioru.

Głównym elementem algorytmu jest tzw. proces „relaksacji" dla każdego wierzchołka sąsiadującego z badanym. Jest to sprawdzenie, czy odległość pomiędzy sąsiadem (`v`) badanego węzła `u` a źródłowym jest większa od sumy odległości między źródłowym a badanym i odległości pomiędzy badanym a jego sąsiadem. Jeśli tak jest, to znaczy, że algorytm znalazł „krótszą" ścieżkę i należy zaktualizować tablice dystansu `d` oraz poprzedników `p`.

Istnieje kilka odmian implementacji algorytmu Dijkstry. Najprostsza wykorzystuje tablicę do przechowywania wierzchołków ze zbioru Q. Inne wersje algorytmu używają kolejki priorytetowej lub kopca Fibonacciego. Przy implementacji bez użycia kopca, złożoność obliczeniowa wynosi: $O(n^2)$, dzięki jego zastosowaniu może spaść do $O(n \cdot log_{10}(n))$[2].

2. Realizacja biblioteki

Do zrealizowania postawionych w pracy celów, zaprojektowano oraz zaimplementowano bibliotekę nazwaną *Simple C Graph Library*, dalej określaną akronimem *SCGL*. Projekt ten stworzony został w oparciu o język C oraz jego bibliotekę standardową (w systemach Unix/Linux: *GNU libc - glibc*). Postanowiono również, że biblioteka będzie implementować algorytm Dijkstry — najkrótszych ścieżek — ze względu na jego popularność oraz istotność dla działania sieci komputerowych (w tym internetu).

Wyboru tego języka programowania dokonano przede wszystkim ze względu na możliwość redukcji wszelkich narzutów wynikających z cech charakterystycznych dla języków obiektowych (dziedziczenie, polimorfizm, szablony). Dodatkowym atutem było bardzo dobre wsparcie kompilatorów oraz szeroki wybór dostępnych narzędzi.

Projekty takie jak ten, często charakteryzują się dynamicznym rozwojem, zwłaszcza w początkowych fazach tworzenia. W celu zapewnienia poprawności zaimplementowanych już funkcjonalności, zdecydowano się skorzystać z mechanizmu testów jednostkowych oraz platformy *DejaGNU*.

Dodatkowo biblioteka wykorzystuje program *make* oraz pliki reguł *Makefile* do automatyzacji procesu kompilacji.

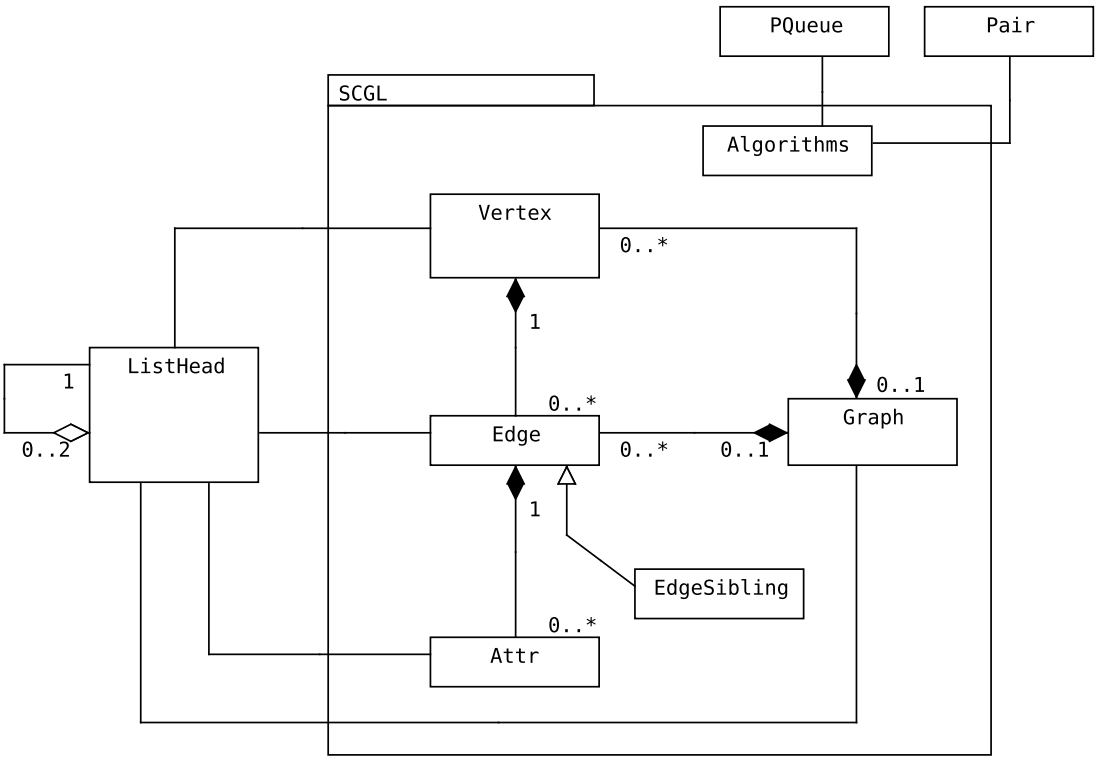
Podczas projektowania każdego z modułów biblioteki wykorzystano wiedzę zawartą w publikacjach ([5], [6]) oraz stosowano się do reguły KISS (ang. *Keep It Simple, Stupid*), która traktuje o tym, że im coś jest prostsze (jako koncept, oraz jako wykonanie) tym lepiej ([5]).

2.1. Budowa projektu

Rozdział ten ma za zadanie przybliżenie czytelnikowi pomysłów projektowych biblioteki SCGL. Zawiera on opis zastosowanej architektury oraz struktury plików.

2.1.1. Diagramy klas

Diagramy, przedstawione w tej części pracy, mają na celu jedynie przekazanie informacji na temat zamysłów projektowych. Poprzez ilustracje struktur klas i zależności między nimi, ukazują one system (bibliotekę SCGL) w modelu obiektowym. Ponieważ do implementacji projektu wybrano język w pełni strukturalny (C) nie było możliwe dokładne odwzorowanie diagramów UML[1].



Rysunek 2. Diagram relacji pomiędzy klasami biblioteki SCGL

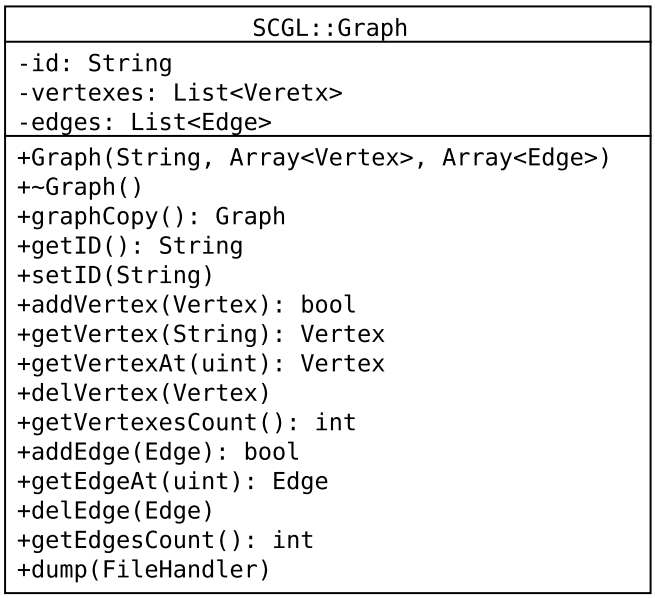
Klasy otoczone pakietem SCGL (ramka) należą do przestrzeni nazw SCGL. Pozostałe klasy są jedynie dodatkiem, lub kodem spoza biblioteki (nie stworzonym przez autora pracy).

SCGL::Graph Klasa rdzeń, najważniejsza w bibliotece SCLG. Modeluje pojęcie grafu przechowując listy wierzchołków oraz krawędzi z nimi związanych. Wykorzystuje do

[1]UML — (ang. *Unified Modeling Language* język formalny wykorzystywany do modelowania różnego rodzaju systemów.

tego celu klasę `List_Head`. Dodatkowo posiada identyfikator `ID`, który jest ciągiem znaków.

Dostarcza metody pozwalające na manipulowanie zawartością swoich list węzłów i krawędzi. Dodatkowo umożliwia wypisanie zawartości grafu na standardowe wyjście (lub plik), a także wykonanie kopii grafu. Funkcja kopiująca, powiela wszystkie elementy grafu, również odwzorowuje powiązania pomiędzy węzłami.



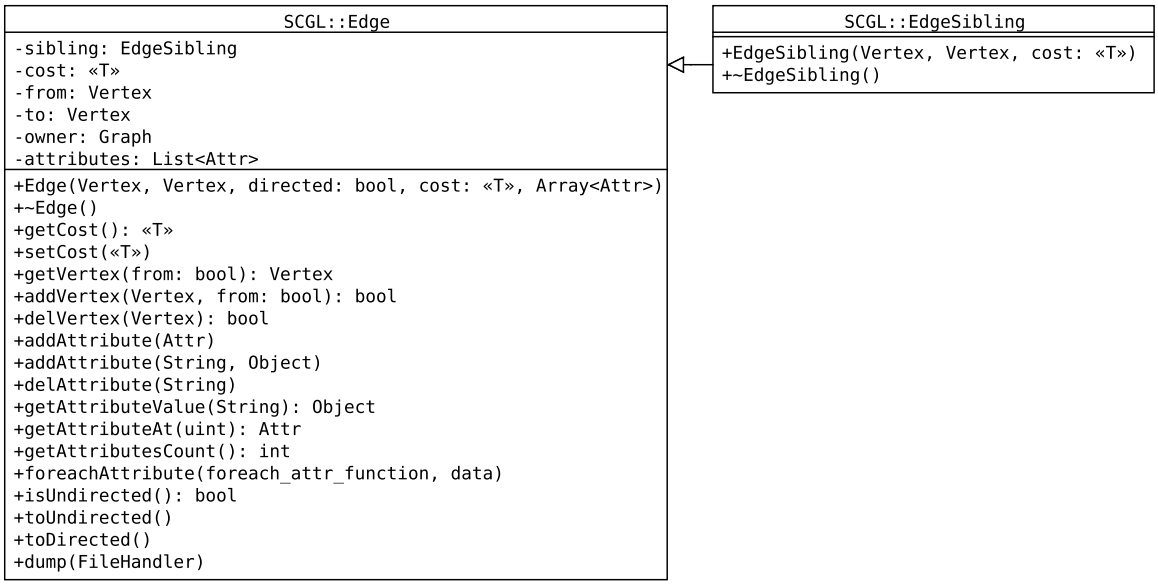
Rysunek 3. Diagram klasy SCGL::Graph

SCGL::Edge Jedna z ważniejszych klas SCGL, reprezentuje krawędź wewnątrz grafu. Określana jest poprzez referencje do dwóch wierzchołków (od, do) oraz grafu, którego jest elementem składowym (tzw. „rodzica"). Dodatkowo posiada dynamiczną listę atrybutów (obiekty klasy SCGL::Attr) oraz generyczne pole koszt (zmienna typu określanego podczas kompilacji).

SCGL::Edge przenosi również informację o jej „rodzeństwie". W przypadku gdy krawędź jest nieskierowana, tworzony jest jej „brat" (obiekt klasy dziedziczącej SCGL::EdgeSibling). Obiekt ten poprzez fakt dziedziczenia, jest klasą zawierającą dokładnie te same składowe co klasa bazowa. Jedyną różnicą jest konstruktor, który nie przyjmuje informacji o atrybutach ponieważ nie ma potrzeby powielania tych

danych. Wierzchołki rodzeństwa są zamienione tj. wierzchołek „od” krawędzi oryginalnej, jest wierzchołkiem „do” brata.

Klasa ta dostarcza metody dostępowe do swych pól prywatnych a także metodę która pozwala na wywołanie funkcji użytkownika na każdym z atrybutów krawędzi. Dodatkowo jak wcześniej opisane klasy, umożliwia również wyprowadzenie jej zawartości na strumień.



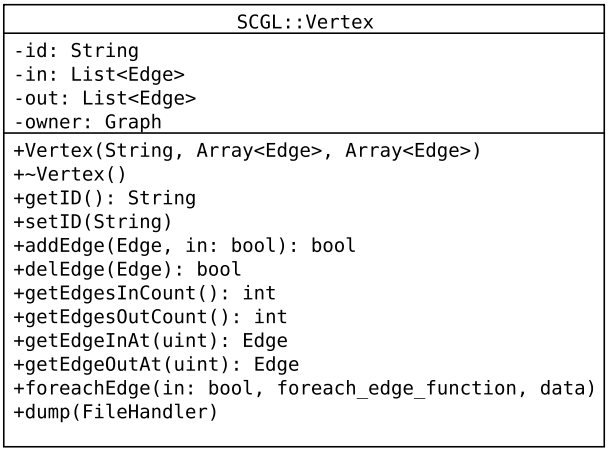
Rysunek 4. Diagram klasy SCGL::Edge

SCGL::Vertex Klasa ta reprezentuje wierzchołek grafu. Podobnie jak SCGL::Graph posiada identyfikator `ID`, który jest ciągiem znaków, oraz metody pozwalające na manipulacje nim. Przechowuje ona dwie listy krawędzi z nią związanych:

- `in` — wchodzących do węzła,
- `out` — wychodzących z węzła.

Jednym z pól klasy jest również referencja do grafu (jej rodzica).

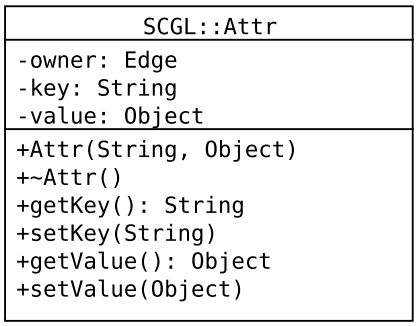
Obiekt tego typu, poza metodami dostępowymi, umożliwia wywołanie funkcji użytkownika na każdej powiązanej z nią krawędzi, jak również wypisanie zawartości na strumień (plik lub standardowe wyjście).



Rysunek 5. Diagram klasy SCGL::Vertex

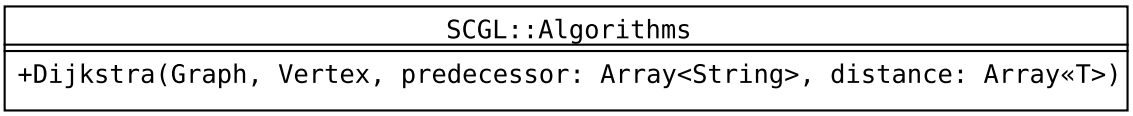
SCGL::Attr Klasa wykorzystywana przez obiekty SCGL::Edge, do przechowywania informacji o jej atrybutach. Opisują ją dwa pola: klucz oraz wartość. Klucz jest ciągiem znaków, za którego unikalność odpowiedzialny jest użytkownik (nie jest to wymagane). Wartość zaś jest referencją do zmiennej dowolnego typu, może to być również struktura danych użytkownika.

Klasa ta udostępnia jedynie konstruktor, destruktor oraz metody dostępowe.



Rysunek 6. Diagram klasy SCGL::Attr

SCGL::Algorithms Jest to klasa, która dostarcza jedynie implementacje algorytmów teorii grafów. Aktualnie zawiera jedynie funkcję obsługującą algorytm Dijkstry, podczas działania którego wykorzystuje klasy Pair oraz PQueue.



Rysunek 7. Diagram klasy SCGL::Algorithms

ListHead Klasa implementująca listę powiązaną dwukierunkową, cykliczną. Całkowicie ukryta przed użytkownikiem biblioteki SCGL, wykorzystywana do wewnętrznych operacji.

PQueue Klasa implementująca kolejkę priorytetową opartą na kopcu. Również niedostępna dla użytkownika, wykorzystywana przez funkcję `Dijkstra` z klasy SCGL::Algorithms.

Pair Klasa implementująca kolejkę parę wartości (identyfikator, dystans) Wykorzystywana przez funkcję `Dijkstra` z klasy SCGL::Algorithms, jako element kolejki priorytetowej.

2.1.2. Struktura plików

Całość projektu: kod źródłowy biblioteki, platformę testów jednostkowych, oraz dodatkowe elementy wykorzystane podczas pisania niniejszej pracy, przechowywana jest przy wykorzystaniu repozytorium *git*[2]. Owe repozytorium umieszczone jest na serwerach darmowego serwisu `https://github.com/`, a jego lokalna kopia na płycie CD dołączonej do pracy (więcej na str. 67).

Zgodnie z dobrą praktyką programistyczną, oraz w celu uporządkowania plików źródłowych, zastosowana została standardowa hierarchia plików. W katalogu głównym `scgl/` (w pliku `LICENSE`) znaleźć można treść licencji (GPL — *General Public License*) biblioteki, krótki opis projektu (`README`) oraz plik ułatwiający m. in. kompilację kodu `Makefile` (więcej na str. 44). Kod źródłowy poszczególnych modułów SCGL znajduje się w podkatalogu `src/`, a pliki definicji struktur oraz funkcje użytkownika (*API*), w podkatalogu `include/`.

[2]Git — rozproszony system kontroli wersji, stworzony przez Linusa Torvaldsa jako narzędzie wspomagające rozwój jądra Linux.

Dodatkowo wewnątrz katalogu `scgl/` znajdziemy pliki stworzone przez generator dokumentacji — *doxygen* (`doc/latex`). Są to pliki tworzące dokumentację interfejsu użytkownika, przy wykorzystaniu komentarzy zawartych w kodzie źródłowym SCGL.

Katalog `unit_tests/scgl.test` zawiera definicje testów *DejaGNU* oraz sam moduł wykorzystywany do testowania biblioteki.

W folderze `perf_tests` znajdują się kody źródłowe (oraz plik Makefile) testów wydajności wykorzystanych do porównania, opisanego w rodziale 3.

2.2. Szczegóły implementacji

Do implementacji przedstawionych wcześniej diagramów, zależności oraz własności projektowych wykorzystano język C. Jest to język strukturalny, nie posiadający cech obiektowości, brak jest tu m. in. dziedziczenia, polimorfizmu, szablonów. Wymusiło to odseparowanie projektu biblioteki od jej ściślej implementacji.

SCGL nie posiada wewnątrz struktur pól prywatnych, wszystko jest dostępne dla użytkownika, jednak nie zalecane jest odwoływanie się do nich bezpośrednio. W przyszłości planowane jest jednak ukrycie owych definicji struktur, dlatego zaleca się korzystać z nazw alternatywnych `typedef`.

Język C nie posiada „klas" przez co nie można za jego pomocą wywoływać metod na rzecz obiektów (jak to ma miejsce w języku C++ np. `my_edge::getID()`). Stworzono zatem specjalny system nazw (przestrzeń nazw) opisany dokładniej w rozdziale 2.3.1.

Pozostałe detale implementacyjne zostały opisane w dalszych podrozdziałach pracy. Wybrano elementy najistotniejsze oraz najciekawsze z punktu działania biblioteki.

2.2.1. Linux Kernel List

Zjawiska modelowane przy wykorzystaniu teorii grafów, charakteryzują się zazwyczaj dużą dynamiką zmian w czasie. Przykładowo, w miarę rozwoju firmy na rynku, rozwija się jej infrastruktura wewnętrzna — struktura sieci komputerowej jest rozbudowywana o nowe lokacje, co za tym idzie urządzenia trasujące (przedstawiane jako węzły grafu). Fakt ciągłych zmian w budowie grafów, narzuca niejako implementującej go bibliotece, wymaganie obsługi tego typu zdarzeń. Musi być ona w stanie dynamicznie zmienić rozmiary struktur, tak aby w każdej chwili dodać (lub usunąć) wybrane elementy.

W językach takich jak *C++* do tego celu wykorzystywane są najczęściej tzw. wektory, czyli tablice o dynamicznych rozmiarach. Korzystając z nich nie musimy podawać, podczas tworzenia, ilości elementów jakie będą w nich przechowywane. A w trakcie dodawania/usuwania elementów, rozmiar tablicy dostosowuje się automatycznie.

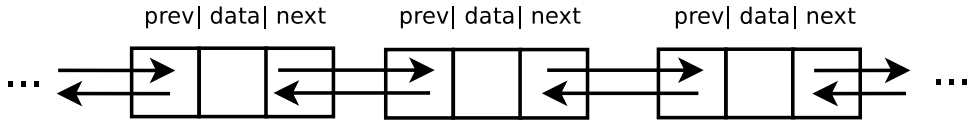
Wybrany do realizacji celów pracy język C, nie posiada wbudowanej obsługi podobnych mechanizmów wbudowanych w standardową bibliotekę. Istnieje co prawda możliwość rozszerzania rozmiaru tablicy (przy pomocy funkcji `realloc`), jednakże jest to mało wydajny mechanizm, zwłaszcza przy dużej ilości operacji dodawania/usuwania. Innym rozwiązaniem byłyby biblioteki zewnętrze, przeznaczone dla języka C, dodające brakującą funkcjonalność. Użycie ich może jednak wiązać się z dodatkowymi dużymi narzutami na rozmiar biblioteki, zużywaną pamięć lub szybkość działania. Aby zniwelować wpływ (negatywny) kodu „trzeciego" na SCGL, postanowiono wykorzystać mechanizm list dowiązanych do przechowywania informacji na temat wszystkich krawędzi (ich atrybutów) oraz węzłów w grafie. Zasada działania jak i implementacja tego typu list jest niezwykle prosta, a jednocześnie nie wpływa znacząco na ilość zużywanej pamięci.

Według klasycznego podejścia do problemu, lista jest to obiekt (struktura/klasa) zawierająca dane właściwie, oraz wskaźnik na kolejny obiekt tego samego typu.[7]

Listing 2.1. Idea listy powiązanej w C

```
1 struct list {
2   struct list *prev
3   struct list *next;
4   void *data;
5 };
```

Posiadając wskaźnik, wskazujący na pierwszy element listy (`head`), możemy otrzymać dostęp do kolejnych danych przechowywanych wewnątrz listy poprzez wyłuskiwanie kolejnych wskaźników (`next`/`prev`).



Rysunek 8. Kolejne elementy listy powiązanej

W niniejszej pracy postanowiono jednak skorzystać z mechanizmu nazwanego *Linux Kernel List*. Jest to implementacja listy dowiązanej, (dwukierunkowej; cyrkulacyjnej),

wykorzystywana w jądrze (ang. *kernel*) systemów operacyjnych *Linux*[10]. Jest to jeden plik nagłówkowy (`*.h`) języka C. Zawiera on definicję struktury `list_head` oraz definicje i deklaracje funkcji oraz makr preprocesora, obsługujących tę strukturę.

Linux Kernel Lists są wyjątkowe, z powodu odmiennego podejścia do tematu list. Struktura `list_head` zawiera jedynie informację (wskaźniki) na następny oraz poprzedni element listy. Brak jest tu standardowego pola `data`, które przechowywałoby informacje użytkownika (tak jak jest to realizowane w standardowej implementacji listy). W rozwiązaniu Linuksowym, realizacja listy daje złudzenie, że lista zawarta jest w obiekcie który łączy (który powinien być wewnątrz niej). Na przykład, jeśli chce się stworzyć listę powiązaną struktur `my_struct`, należy zrobić to w następujący sposób:

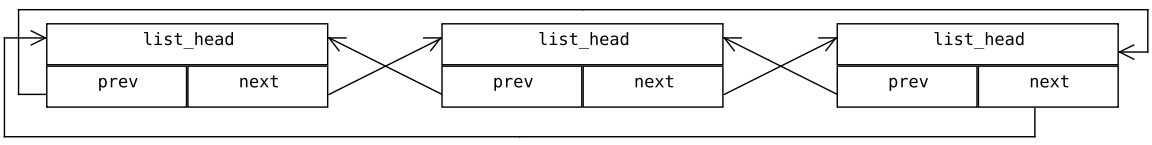
Listing 2.2. Przykład tworzenia listy powiązanej za pomocą Kernel Lined List

```
1 struct my_struct{
2   struct list_head list; /* struktura kernel list */
3   int my_data;
4   void *my_void;
5 };
```

Gdzie struktura `list_head` zbudowana jest następująco:

```
1 struct list_head{
2   struct list_head *prev;
3   struct list_head *next;
4 };
```

Dołączamy strukturę listy do własnego obiektu, zamiast, obiekt do struktury listy. Dodając kolejne elementy listy, łączymy ze sobą tak naprawdę kolejne struktury `list_head`.



Rysunek 9. Kolejne elementy listy powiązanej w stylu Linux Kernel List

Bardzo ciekawym mechanizmem jest makro `container_of` pozwalające wyznaczyć początek struktury przy pomocy jednego z jej elementów składowych.

Listing 2.3. Definicja jednego z makr jądra linuksa: container_of

```
#define container_of(ptr, type, member) \
   ((type *)((char *)(ptr)-(unsigned long)(&((type *)0)->member)))
```

Otrzymując kolejno: wskaźnik na jedno z pól struktury, typ struktury, oraz nazwę tego pola; odejmuje offset członka struktury (od jej początku) od adresu wskaźnika, dzięki czemu otrzymuje i zwraca adres początku struktury[11]. Makro to wykorzystywane jest podczas poruszania się po elementach list, do zwracania adresów struktur, podczas gdy mamy dany tylko wskaźnik na `list_head_t`.

Najważniejsze przy obcowaniu z listami jądra Linuksa to:

- Lista jest wewnątrz obiektów, które chcemy razem połączyć.
- Można umieścić strukturę `list_head` w każdym miejscu własnej struktury.
- Zmienna typu `list_head` może mieć dowolną nazwę.
- Rozwiązanie to nie ogranicza nas do jednej listy, na cały kod źródłowy.

Ponieważ jest to mechanizm zapożyczony z jądra systemu Linux, istnieje pewność iż jest dobrze przetestowany, przenośny, szybki oraz zajmuje niewiele pamięci. Warto również wspomnieć, iż ten sam plik nagłówkowy, dostarcza podstaw do implementacji tablic asocjacyjnych (ang. *hash list*).

Potrzebne były pewne modyfikacje, aby plik nagłówkowy `list.h` dostosować do zwykłego kodu, przestrzeni użytkownika (ang. *userspace*):

1. Została zmieniona definicja pliku nagłówkowego `list.h`
2. Zostały usunięte załączone pliki nagłówkowe
3. Zostały dodane struktury znajdujące się w pliku `types.h`
4. Wywołanie makra `offsetof` z pliku `stddef.h`, zostało zamienione na jego treść
5. Zmienne `LIST_POISON1` oraz `LIST_POISON2` zostały zastąpione przez wskaźnik na NULL — sens pozostaje taki sam

Dodatkowo na potrzeby biblioteki SCGL dopisana została funkcja `list_count` obliczająca ilość elementów wewnątrz listy:

Listing 2.4. Ciało funkcji list_count

```
static inline unsigned int
list_count(const struct list_head *head) {
  unsigned int i = 0;
  struct list_head *j;
  list_for_each(j, head) {
    ++i;
  }
  return i;
}
```

Jak już zostało wspomniane, łącząc kolejne elementy, łączymy ze sobą struktury `list_head`. Wymusza to na programiście, zmianę sposobu myślenia. Chcąc przechowywać listę krawędzi wewnątrz struktury grafu, musimy w obu tych obiektach dołączyć strukturę `list_head`. Im więcej list krawędzi chce się stworzyć, tym więcej zmiennych typu `list_head` należy wpisać w budowę obiektu.

I tak biblioteka SCGL posiadając pięć list:

1. krawędzi w grafie,
2. węzłów w grafie,
3. krawędzi wchodzących do węzła,
4. krawędzi wychodzących z węzła,
5. atrybutów krawędzi.

Musi mieć wpisane w struktury dziesięć, zmiennych typu `list_head`.

Listing 2.5. Zastosowanie Linux Kernel List na przykładzie struktur biblioteki SCGL

```
/** attribute object */
struct scgl_attr {
  list_head_t list;
};

/** edge object */
struct scgl_edge {
  list_head_t from_list;
  list_head_t to_list;
  list_head_t owner_list;
  list_head_t attributes;
};
```

```
13
14 /** vertex object */
15 struct scgl_vertex {
16   list_head_t in;
17   list_head_t out;
18   list_head_t owner_list;
19 };
20
21 /** graph object */
22 struct scgl_graph {
23   list_head_t vertexes;
24   list_head_t edges;
25 };
```

Taka metoda przechowywania informacji przypomina nieco listy sąsiedztwa (czyt. na str. 12), nie jest jednak dokładną ich implementacją.

Zastosowanie list powiązanych, zapożyczonych z jądra Linuksa, pozwoliło na oszczędzenie zużywanej przez struktury pamięci oraz wzrost szybkości działania. Przyśpieszyło i ułatwiło to również sam proces tworzenia biblioteki SCGL, gdyż nie warto tworzyć kolejnych rozwiązań od nowa, jeśli istniejące są dobrze zaimplementowane.

2.2.2. Statycznie kompilowany typ zmiennej

Sporym oraz ciekawym wyzwaniem projektowym okazał się niepozorny atrybut kosztu (wagi) krawędzi, a właściwie typ zmiennej przechowującej tą wartość. Jak już zostało wcześniej wspomniane, element ten wykorzystywany jest przy wyborze ścieżki pomiędzy zadanymi węzłami. Wymaganiem postawionym przed biblioteką SCGL była elastyczność typu zmiennej określającej koszt krawędzi. Powinna ona pozwalać użytkownikowi na wybór typu owego parametru według własnych preferencji, bez ingerencji w sam kod projektu.

Problem ten dotyka tematyki paradygmatu programowania uogólnionego (generycznego, ang, *generic programming*). Pozwala on na pisanie kodu programu bez wcześniejszej znajomości typów danych, na których kod ten będzie pracował. W językach Java, C#, Haskell służą do tego typy generyczne (typ ten pojawia się również w C++ dzięki zastosowaniu biblioteki *boost::any*). Zaś w językach takich jak C++ czy D, funkcjonalność tą można zrealizować poprzez zastosowanie mechanizmu szablonów (ang. *template*). Podczas kompilacji następuje tak zwana konkretyzacja szablonu (ang. *template instantiation*), podczas której kompilator na podstawie typów danych przekazanych wzorcowi generuje

kod właściwy do obsługi danego typu. Dla każdego użycia szablonu z innym typem, generowana jest kopia odpowiednich fragmentów kodu.

Wybrany dla biblioteki SCGL język C nie posiada żadnego z wyżej wymienionych mechanizmów. Aby spełnić postawione wymagania rozważano kilka możliwości projektowych, jedną z nich było zastosowanie unii z kilkoma podstawowymi typami zmiennych skalarnych, oraz dodatkowego pola określającego wybrany typ. Wykorzystanie unii do tego celu, miałoby ograniczyć ilość zużytej pamięci, do największej zmiennej wewnątrz unii.

Listing 2.6. Koszt krawędzi jako unia

```
1  enum cost_type {INT, DOUBLE, FLOAT};
2
3  struct scgl_edge {
4     union cost {
5          int i;
6          double d;
7          float f;
8     };
9     cost_type type;
10 };
```

Rozwiązanie to niestety wymagałoby każdorazowego sprawdzania wartości zmiennej `type` przed użyciem zmiennej `cost` (ponieważ należy wskazać którą zmienną wybieramy z unii). Instrukcja warunkowa `switch` (rozrastająca się w miarę dodawania nowych typów do unii) negatywnie wpłynęłaby na wydajność funkcji wykorzystujących atrybut kosztu. Alternatywnym rozwiązaniem byłoby stworzenie, dla każdego typu, funkcji posługujących się odpowiednią zmienną z unii np:

```
void scgl_dijsktra(/*...*/)
void scgl_dijkstra_int(/*...*/)
void scgl_dijkstra_float(/*...*/)
void scgl_dijkstra_double(/*...*/)
```

Zmniejszyłoby to narzut wynikający z każdorazowego sprawdzania typu zmiennej kosztu (sprawdzanie odbywałoby się przez warper `scgl_dijkstra()`, który wywoływałby odpowiednią funkcję). Niestety rozwiązanie to jednocześnie zwiększyłoby znacznie rozmiar biblioteki, dodatkowo narażając kod na błędy rodzaju copy-paste (wynikające z powielania ciała funkcji poprzez kopiowanie i wklejanie). Dodatkową wadą tego rozwiązania, jest fakt, iż unia zajmuje tyle miejsca co jej największa składowa. Gdyby w jej wnętrzu znala-

zła się zmienna typu `long double` to mimo iż programista wykorzystywał by `cost` jako zmienną short, to pole zajmowałoby tyle bajtów ile `long double` na danej architekturze.

Z powodu wyżej wymienionych cech, zdecydowano się zastosować zupełnie inne rozwiązanie. Postanowiono stworzyć wewnątrz pliku nagłówkowego *scgl_edge.h* alternatywną nazwę (`typedef`), a następnie zdefiniować wewnątrz struktury krawędzi pole będące realizacją jej kosztu w następujący sposób:

Listing 2.7. Koszt krawędzi w bibliotece SCGL

```
1 typedef cost_type cost_type_t;
2
3 struct scgl_edge {
4   cost_type_t cost;
5 };
```

Kod ten oczywiście nie ma prawa zadziałać, gdyż symbol `cost_type` dalej pozostaje niezdefiniowany. Odpowiedzialność za tę czynność przeniesiono do etapu kompilacji biblioteki, a właściwie etapu translacji. Kompilator *gcc* poprzez opcję `-D` umożliwia definiowanie nazw, traktując je tak jakby w kodzie pojawił się odpowiednio skonstruowany blok `#define`.

```
gcc -Dname=definition
gcc -Dcost_type=double
gcc -Dcost_type=int
```

Kompilując kod z tym parametrem słowo `cost_type` jest podmieniane na wybrane przez użytkownika, co za tym idzie zmienna `cost` przybiera pożądany typ.

Statyczny typ zmiennej, dobierany podczas procesu kompilacji nie powoduje dodatkowego narzutu przed jej użyciem (jak to miało miejsce w przypadku unii), czy na rozmiar biblioteki (szablony języka C++). Wadą jest tu jednak potrzeba rekompilacji całej biblioteki (wszystkich modułów, które korzystają ze zmiennej), za każdym razem gdy użytkownik zechce zmienić jej typ. Jest to jednak niedogodność, którą można zaakceptować, zważywszy na możliwości, którą oferuje owe rozwiązanie.

Dodatkowym elementem wynikającym z wybranego rozwiązania, jest potrzeba dbania o zależności. Wybór typu zmiennej powinien dostarczać dodatkowych informacji takich jak minimalna/maksymalna wartość zmiennej, oraz format zmiennej rozumiany przez funkcję `printf`. Aby ułatwić proces budowania biblioteki, oraz zmniejszyć ewentualną możliwość popełnienia błędu przez użytkownika, postanowiono, że doborem wcześniej wymienionych

wartości zajmie się plik *Makefile*. Wewnątrz tego pliku zdefiniowane są reguły budowania całej biblioteki (oraz dodatkowych elementów). Makefile definiuje zmienną `COST_TYPE`, która może przyjąć wartości odpowiadające określonemu typowi zmiennej `cost`. Na podstawie wyboru użytkownika (zmiennej `COST_TYPE`), Makefile dobierze odpowiednie opcje tak aby biblioteka została skompilowana z obsługą kosztu krawędzi o wybranym typie. W tym celu wewnątrz pliku Makefile stworzono blok decydujący o następującej treści:

Listing 2.8. Makefile - blok decydujący o zmiennej cost

```
ifneq (,$(findstring s,$(COST_TYPE)))
  override MFLAGS:=-Dcost_type="short" -Dcost_fmt=\"%hd\" -Dcost_max=SHRT_MAX -Dcost_min=
      SHRT_MIN
endif
ifneq (,$(findstring us,$(COST_TYPE)))
  override MFLAGS:=-Dcost_type="unsigned short" -Dcost_fmt=\"%hd\" -Dcost_max=USHRT_MAX -
      Dcost_min=USHRT_MIN
endif
ifneq (,$(findstring i,$(COST_TYPE)))
  override MFLAGS:=-Dcost_type="int" -Dcost_fmt=\"%d\" -Dcost_max=INT_MAX -Dcost_min=
      INT_MIN
endif
ifneq (,$(findstring ui,$(COST_TYPE)))
  override MFLAGS:=-Dcost_type="unsigned int" -Dcost_fmt=\"%d\" -Dcost_max=UINT_MAX -
      Dcost_min=UINT_MIN
endif
ifneq (,$(findstring l,$(COST_TYPE)))
  override MFLAGS:=-Dcost_type="long" -Dcost_fmt=\"%ld\" -Dcost_max=LONG_MAX -Dcost_min=
      LONG_MIN
endif
ifneq (,$(findstring ul,$(COST_TYPE)))
  override MFLAGS:=-Dcost_type="unsigned long" -Dcost_fmt=\"%ld\" -Dcost_max=ULONG_MAX -
      Dcost_min=ULONG_MIN
endif
ifneq (,$(findstring ll,$(COST_TYPE)))
  override MFLAGS:=-Dcost_type="long long" -Dcost_fmt=\"%lld\" -Dcost_max=LLONG_MAX -
      Dcost_min=LLONG_MIN
endif
ifneq (,$(findstring ull,$(COST_TYPE)))
  override MFLAGS:=-Dcost_type="unsigned long long" -Dcost_fmt=\"%lld\" -Dcost_max=
      ULLONG_MAX -Dcost_min=ULLONG_MIN
endif
ifneq (,$(findstring f,$(COST_TYPE)))
  override MFLAGS:=-Dcost_type="float" -Dcost_fmt=\"%f\" -Dcost_max=FLT_MAX -Dcost_min=
      FLT_MIN
endif
ifneq (,$(findstring d,$(COST_TYPE)))
```

```
29   override MFLAGS:=-Dcost_type="double" -Dcost_fmt=\"%f\" -Dcost_max=DBL_MAX -Dcost_min=
         DBL_MIN
30 endif
31 ifneq (,$(findstring ld,$(COST_TYPE)))
32   override MFLAGS:=-Dcost_type="long double" -Dcost_fmt=\"%Lf\" -Dcost_max=LDBL_MAX -
         Dcost_min=LDBL_MIN
33 endif
```

Przedstawiony powyżej wycinek pliku Makefile porównuje zawartość zmiennej `COST_TYPE` z ustalonymi wcześniej wartościami. Na tej podstawie dobiera odpowiednie zależności, przedstawione w tabeli 1.

Tablica 1. Zależność poszczególnych zmiennych od wartości `COST_TYPE`

COST_TYPE	cost_type	cost_fmt	cost_max	cost_min
s	short	%hd	SHRT_MAX	SHRT_MIN
us	unsigned short	%hd	USHRT_MAX	USHRT_MIN
i	int	%d	INT_MAX	INT_MIN
us	unsigned int	%d	UINT_MAX	UINT_MIN
l	long	%ld	LONG_MAX	LONG_MIN
ul	unsigned long	%ld	ULONG_MAX	ULONG_MIN
ll	long long	%lld	LLONG_MAX	LLONG_MIN
ull	unsigned long long	%lld	ULLONG_MAX	ULLONG_MIN
f	float	%f	FLT_MAX	FLT_MIN
d	double	%f	DBL_MAX	DBL_MIN
ld	long double	%Lf	LDBL_MAX	LDBL_MIN

Funkcja `findstring` wyszukuje wystąpienia pierwszego argumentu wewnątrz drugiego argumentu. Jeśli wyszukiwanie zakończyło się sukcesem, funkcja zwraca znaleziony ciąg znaków, w przeciwnym wypadku wynik funkcji jest równy pustemu ciągowi (dlatego sprawdzany jest warunek `not equal` z pustym pierwszym argumentem).

Posługiwanie się zmienną `cost` odbywa się dokładnie tak jak innymi zmiennymi typu skalarnego w języku C. Zalecane, aczkolwiek nie wymagane, jest korzystanie z definicji `cost_max`, `cost_min` oraz `cost_fmt`. Wartości `cost_max` oraz `cost_min` zdefiniowane są w plikach nagłówkowych *limits.h* oraz *float.h*. Ich załączenie wymagane jest w każdym pliku `*.c` odwołującym się do zmiennej o typie `cost_type`.

Użytkownik chcący dodać własny typ może dokonać edycji pliku Makefile. Powinien przypisać nazwę typu do deklaracji `cost_type`, jej wartość minimalną do `cost_min`, maksymalną do `cost_max`, oraz format dla funkcji `printf` do `cost_fmt`.

Listing 2.9. Przykład nowego typu kosztu krawędzi

```
ifneq (,$(findstring nowy,$(COST_TYPE)))
  override MFLAGS:=-Dcost_type=nowy -Dcost_fmt=\"%format\" -Dcost_max=99 -Dcost_min=-99
endif
```

Należy pamiętać o modyfikowaniu znaków (ang. *characters escaping*) przy definiowaniu wartości `cost_fmt`.

Dzięki zaproponowanemu rozwiązaniu, biblioteka SCGL zdecydowanie zyskuje na wydajności zarówno czasowej jak i pamięciowej, nie wprowadzając przy tym zbytniej komplikacji kodu.

2.2.3. Algorytm Dijkstry

Do implementacji jednego z algorytmów teorii grafów wybrano algorytm Dijkstry. Wyboru tego dokonano ze względu na jego wydajność oraz istotny udział w rozwoju sieci komputerowych, które są tematem przewodnim studiów autora pracy.

W opisie teoretycznym algorytmu (rozdział 1.1) wspomniano, że złożoność czasowa algorytmu zależy od wykorzystanej metody przechowywania węzłów. Aby rozwiązanie dostarczone przez bibliotekę SCGL, było możliwie jak najszybsze, zdecydowano się na użycie kolejki priorytetowej opartej na kopcu. Złożoność algorytmu powinna wówczas wynosić $O(n \cdot log_{10}(n))$[8]. Kolejka ta powinna priorytetować węzły na podstawie ich odległości od węzła źródłowego, oraz dostarczać operacje wyciągania elementu o najwyższym priorytecie (najniższej odległości) oraz modyfikowania priorytetu (tzw. *decrease-key*).

Postanowiono wykorzystać gotową implementację, stworzoną przez *Andrei Ciobanu* `http://andreinc.net/2011/06/01/`. Rozwiązanie to udostępnione jest publicznie, bez licencji, a autor kodu wyraził zgodę na użycie go w bibliotece SCGL:

> Yes, you can freely use the priority queue implementation. What I recommend you to do is to see test it very good to see if it's not bugged. Good luck with your thesis.
>
> Andrei

Kod ten zmieniono pod względem stylistycznym, tak aby pasował do reszty biblioteki SCGL, dopisano funkcję `pqueue_replace_data`, która realizuje operację zmiany priorytetu, oraz poprawiono kilka błędów logicznych.

Implementacja algorytmu Dijkstry napisana dla biblioteki SCGL wzoruje się na rozwiązaniu zastosowanym w Boost::BGL[9]. Stosuje ona kolejkę priorytetową oraz kolorowanie węzłów, czyli operację oznaczania odwiedzonych już elementów.

Listing 2.10. Pseudokod implementowanego w SCGL algorytmu Dijkstry

```
1  for each vertex u in V
2    d[u] := infinity
3    p[u] := u
4    color[u] := WHITE
5  end for
6
7  color[s] := GRAY
8  d[s] := 0
9  INSERT(Q, s)
10 while (Q is not empty)
11   u := EXTRACT-MIN(Q)
12   for each vertex v in Adj[u]
13     if (w(u,v) + d[u] < d[v])
14       d[v] := w(u,v) + d[u]
15     p[v] := u
16     if (color[v] = WHITE)
17       color[v] := GRAY
18       INSERT(Q, v)
19     else if (color[v] = GRAY)
20       DECREASE-KEY(Q, v)
21     end if
22   end for
23   color[u] := BLACK
24 end while
```

Jest to rozwiązanie bardzo optymalne, wykorzystanie mechanizmu kolorowania eliminuje dodatkowe (niepotrzebne) analizy mogące pojawić się w trakcie działania algorytmu. W przypadku napotkania wierzchołka o „kolorze szarym” wywoływana jest operacja zmiany priorytetu tego węzła w kolejce, tak aby był on jak najniższy, wówczas będzie brany on pod uwagę w ostatniej kolejności (w dalszych przejściach pętli). Niestety z powodu zastosowania list dowiązanych w bibliotece SCGL, realna implementacja powyższego algorytmu wymaga ciągłego obliczania jaki numer kolejny posiada określony węzeł. Jest to operacja przechodząca po wszystkich elementach listy, do momentu odnalezienia

określonego wierzchołka, co w przypadku dużych grafów skutkować będzie znacznym spowolnieniem funkcji. Sytuację tą poprawiłoby zastosowanie tablic asocjacyjnych (ang. *hash table*), które jednoznacznie odwzorowywałby adres w pamięci RAM z identyfikatorem węzła.

2.2.4. Testy jednostkowe - DejaGNU

Testy jednostkowe są nieodłączną częścią każdego większego projektu programistycznego. Jest to metoda testowania tworzonego oprogramowania poprzez weryfikowanie poprawności działania, każdego z pojedynczych elementów (jednostek — ang. *units*) programu. Testowany fragment poddawany jest weryfikacji otrzymanego wyniku z oczekiwanym (tak pozytywnym, jak i negatywnym). Testy jednostkowe pomagają weryfikować poprawność funkcji, mimo wprowadzanych zmian. Programista wprowadzając kolejne usprawniania, funkcjonalności, po wykonaniu testów, może mieć pewność, że jego zmiany w kodzie nie wprowadziły kolejnych błędów. Projektując bibliotekę SCGL, postanowiono wykorzystać właśnie tę metodę testowania, aby usprawnić przyszły jej rozwój.

Zdecydowano wykorzystać do tego celu framework DejaGNU. Przeznaczeniem tego oprogramowania, jest stworzenie warstwy abstrakcyjnej dla wszelakich testów[12]. Napisany został z wykorzystaniem pakietu *expect*, który jest częścią języka *Tcl* (ang. *Tool Command Language*). Expect, tworzy własny terminal, który symuluje działanie użytkownika obsługującego testujący program (najczęściej konsolowy)[13]. Pozwala on na dość elastyczną interakcję, wygodne porównywanie oraz reagowanie na wyjście testowanego programu.

Konsekwentnie stosując zasadę *KISS* (czyt. str. 17) postanowiono, że również testy jednostkowe dla biblioteki SCGL będą napisane w sposób jasny i przejrzysty. Platforma testowa składa się z dwóch plików, znajdujących się w katalogu `scgl/unit_test/scgl.test`:

- tests.c — zawierają kody aplikacji/podprogramów wykorzystujących bibliotekę SCGL
- test.exp — definiuje testy, oraz wartości oczekiwane tych testów

Program budowany przy pomocy pliku `tests.c` zawiera 23 podprogramy, każdy z nich korzysta/testuje konkretną „jednostkę" biblioteki SCGL. Program ten oczekuje na wpisanie

przez użytkownika: rodzaju podprogramu (wybór modułu), oraz jego numeru (wybór jednostki), a następnie uruchamia przykładowy kod. DejaGNU, uruchamiając kolejne testy, porównuje wartości oczekiwane, z tymi zwróconymi przez program `tests.out`.

Przy pomocy wyżej opisanych plików, testowane jest:

1. Tworzenie/usuwanie węzłów
2. Ustawianie/pobieranie ID węzła
3. Tworzenie/usuwanie krawędzi
4. Ustawianie/pobieranie kosztu krawędzi
5. Dodawanie/pobieranie/usuwanie węzła do/z krawędzi
6. Dodawanie/pobieranie/usuwanie krawędzi do/z węzła
7. Zliczanie krawędzi w węźle
8. Wykonywanie funkcji na każdej krawędzi w węźle
9. Tworzenie/usuwanie atrybutu krawędzi
10. Ustawianie/pobieranie klucza atrybutu
11. Ustawianie/pobieranie wartości atrybutu
12. Dodawanie/pobieranie/usuwanie atrybutu do/z krawędzi
13. Zliczanie atrybutów krawędzi
14. Wykonywanie funkcji na każdym atrybucie
15. Zmiana krawędzi na (nie)skierowaną
16. Tworzenie/usuwanie grafu
17. Ustawianie/pobieranie ID grafu
18. Dodawanie/pobieranie/usuwanie węzła do/z grafu
19. Dodawanie/pobieranie/usuwanie krawędzi do/z grafu

20. Zliczanie węzłów/krawędzi w grafie

21. Kopiowanie grafów

22. Wykonywanie algorytmu Dijkstry na grafie skierowanym

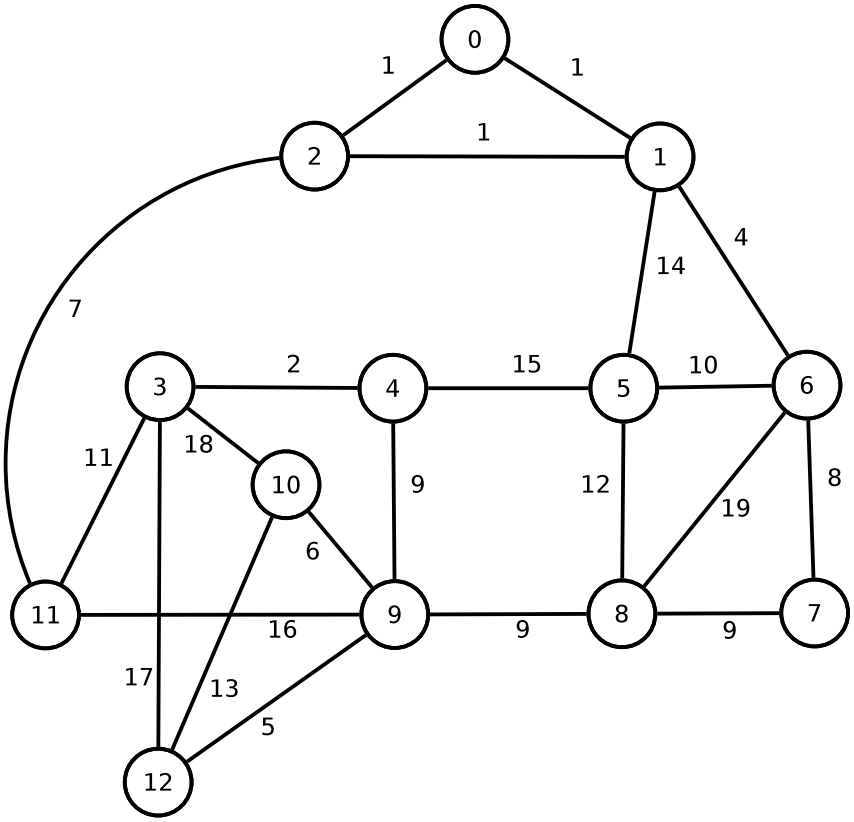
23. Wykonywanie algorytmu Dijkstry na grafie nieskierowanym

Listing 2.11. Plik definicji testów jednostkowych platformy DejaGNU

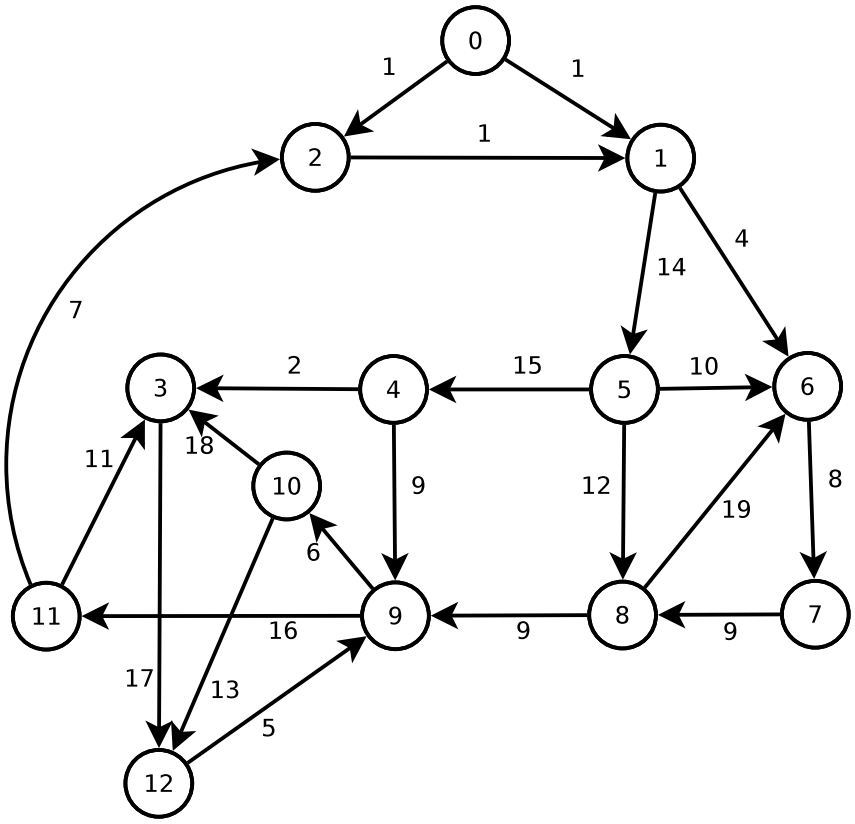
```
global TEST_APP
set test_data {
  {"vertex create/destroy"        "v 1"   "^test 0$"}
  {"vertex set/get ID"            "v 2"   "^test_get test_set$"}
  {"edge create/destroy"          "e 1"   "^0 1 123 123 0$"}
  {"edge get/set cost"            "e 2"   "^123 321$"}
  {"edge add/get/del vertex"      "e 3"   "^\\d+ 0 0 \\d+ 0 \\d+ \\d+ 0$"}
  {"vertex add/get/del edge"      "v 3"   "^\\d+ 0 0 0 \\d+ 0$"}
  {"vertex edges count"           "v 4"   "^0 0 1 0 1 1$"}
  {"vertex foreach edge"          "v 5"   "^1 2 $"}
  {"attribute create/destroy"     "a 1"   "^key value 0$"}
  {"attribute get/set key"        "a 2"   "^key key2.$"}
  {"attribute get/set value"      "a 3"   "^val1 val2.$"}
  {"edge add/get/del attributes"  "e 4"   "^key1 val1 val1 0$"}
  {"edge attributes count         "e 5"   "^1 0$"}
  {"edge foreach attribute"       "e 6"   "^key val.$"}
  {"edge set/get (un)directed"    "e 7"   "^0 123 1 1 \\d+ 123 1$"}
  {"graph create/destroy"         "g 1"   "^G1 0$"}
  {"graph get/set ID"             "g 2"   "^G1 G2.$"}
  {"graph add/get/del vertex"     "g 3"   "^2 V2 V1 1 0 V2.$"}
  {"graph add/get/del edge"       "g 4"   "^2 123 321 1 321 0$"}
  {"graph vertexes/edges count"   "g 5"   "^0 0 2 2$"}
  {"graph copy"                   "g 6"   "^0 1 1 0 1 V2 V2 0 1$"}
  {"dijkstra directed graph       "d 1"
    "^0, 0, 0, 4, 5, 1, 1, 6, 7, 8, 9, 9, 3, 0, 1, 1, 32, 30, 15, 5, 13, 22, 31, 37, 47,
        49,.$"}
  {"dijkstra undirected graph"    "d 2"
    "^0, 0, 0, 11, 3, 1, 1, 6, 7, 11, 9, 2, 9, 0, 1, 1, 19, 21, 15, 5, 13, 22, 24, 30, 8,
         29,.$"}
}

foreach pattern $test_data {
  eval "spawn $TEST_APP [lindex $pattern 1]"
  expect {
    -re [lindex $pattern 2] { pass [lindex $pattern 0] }
    default { fail [lindex $pattern 0] }
  }
```

Plik opisujący procedurę testowania `test.exp`, przechowuje tabelę `test_data` (linie 2–28) zawierającą kolejno: nazwę testu, argumenty przekazywane programowi oraz wartości oczekiwane (zapisane przy pomocy języka wyrażeń regularnych — *regexp*). Część wykonawcza testu, to pętla `foreach` (wiersze 30-35) wywołująca kolejno testy z tabeli `test_data` oraz porównująca wyniki programu z wartościami oczekiwanymi. Testy wywoływane są w kolejności, która zapewnia spójność logiczną całości, np.: test dodawania atrybutu do krawędzi, jest zależy od testu tworzenia krawędzi oraz od testu tworzenia atrybutu, dlatego też jest wywoływany w ostatniej kolejności. Testy algorytmu Dijkstry są przeprowadzane na grafach zaprezentowanych na rysunkach 11, 10.



Rysunek 10. Wykorzystywany do testów graf nieskierowany



Rysunek 11. Wykorzystywany do testów graf skierowany

Wymogiem niezbędnym do uruchomienia testów jest kompilujący się kod biblioteki SCGL, oraz posiadanie zainstalowanego w systemie pakietu DejaGNU. Testy uruchamiane są przez Makefile znajdujący się w katalogu `scgl/`, można dokonać tego w następujący sposób:

```
make tests
```

Polecenie to wywoła kompilację pliku `scg/unit_tests/scgl.test/tests.c` oraz uruchomi program `runtest` (część pakietu DejaGNU) z plikiem definicji testów: `test.exp`.

Jak zostało wcześniej wspomniane, testy jednostkowe są bardzo ważnym elementem oprogramowania, które będzie rozwijane. Należy jednak pamiętać, że w samych testach również może wkraść się błąd, należy je zatem tworzyć ze szczególną starannością.

2.3. Instrukcja użytkownika

Niniejszy rozdział pracy ma na celu przybliżenie użytkownikowi ideologi zastosowanej dla interfejsu API oraz wyjaśnić budowę oraz sposób działania skryptów kompilujących —

Makefile. Bardziej szczegółowe informacje na temat, każdej z funkcji zaimplementowanej w bibliotece SCGL dostarcza, załączona na płycie CD, dokumentacja wygenerowana przez program *Doxygen*.

2.3.1. Interfejs programisty - API

API (ang. *Application Programming Interface*), interfejs programowania aplikacji. Jego zadaniem jest dostarczenie struktur danych oraz funkcji ułatwiających manipulowanie nimi[12].

Biblioteka SCGL, jak wiele innych projektów korzystających z języka C, wykorzystuje system prefiksów do nazewnictwa funkcji. Wszystkie funkcje rozpoczynają się od przestrzeni nazw `scgl_`, następnie występuje nazwa modułu np. `edge_` i właściwa nazwa funkcji np.:

Listing 2.12. Przykłady przestrzeni nazw biblioteki SCGL

```
scgl_edge_add_vertex(...);
scgl_vertex_create(...);
scgl_graph_destroy(...);
```

Definicje obiektów, czyli struktury, również posiadają przedrostek `scgl_`. Użytkownik powinien jednak, korzystać z alternatywnej definicji nazwy typu (`typedef`), czyli typów zakończonych literą „`t`” np. `scgl_edge_t`:

```
scgl_edge_t my_edge;
scgl_graph_t *ptr_my_graph;
```

Docelowo nazwy te powinny być wykorzystywane do ukrycia zawartości poszczególnych struktur (odpowiednik pól prywatnych klas).

Większość, jak nie wszystkie, funkcje w SCGL jako pierwszy argument przyjmują obiekt, który będzie podlegał modyfikacjom, lub który posiada w swojej strukturze wymagane elementy. Argument ten jest odpowiednikiem obiektowego wskaźnika `this`. Wywołanie funkcji:

```
scgl_vertex_get_edges_out_count(v);
```

W języku C++ można by zapisać w następujący sposób:

```
using namespace scgl;
/* ... */
v::get_edges_out_count();
```

Funkcje tworzące i niszczące obiekty

Najważniejszymi funkcjami biblioteki SCGL, są oczywiście te pozwalające na stworzenie oraz zniszczenie obiektów grafowych (grafu, węzła, krawędzi). Funkcje te, we wszystkich modułach, przyjmują te same sufiksy (końcówki nazwy), odpowiednio dla tworzenia `_create` i niszczenia `_destroy`. Przykłady tych funkcji znajdują się na listingu 2.12 w liniach 2-3. Podczas korzystania z tych metod należy pamiętać o:

- Funkcje tworzące alokują potrzebną ilość pamięci zwracając wskaźnik na początek struktury.
- Funkcje niszczące wymagają podania referencji do wskaźnika, dzięki temu po uwolnieniu pamięci jest on zerowany.
- Funkcja niszcząca krawędzie (oraz graf) wymaga podania wskaźnika do funkcji, która będzie potrafiła uwolnić pamięć dla opcjonalnych atrybutów krawędzi.

Atrybuty krawędzi

Podstawowym atrybutem krawędzi w bibliotece SCGL jest jej koszt (`cost`). Istnieje on zawsze, dla każdej krawędzi, inicjowany jest przez użytkownika w trakcie tworzenia obiektu.

Dodatkowo biblioteka przewiduje możliwość wykorzystywania atrybutów definiowanych przez użytkownika. Atrybuty te reprezentowane są przez strukturę `scgl_attr`. Z punktu widzenia użytkownika SCGL, ważne jest, że struktura ta symuluje działanie elementów tablicy asocjacyjnej. Wartość `value` jest powiązana z kluczem `key` (dbanie o niepowtarzalność klucza nie jest wymagana). Każda z krawędzi posiada listę (więcej na str. 23) obiektów `scgl_attr`. Dzięki temu istnieje możliwość przechowywania dowolnej ilości atrybutów w pojedynczej krawędzi (ograniczeniem jest jedynie pamięć RAM oraz SWAP systemu).

Manipulowanie atrybutami z poziomu krawędzi umożliwiają metody:

- `scgl_edge_add_attribute`
- `scgl_edge_add_attribute_object`
- `scgl_edge_del_attribute`

- `scgl_edge_get_attribute_value`
- `scgl_edge_get_attribute_at`
- `scgl_edge_get_attributes_count`
- `scgl_edge_foreach_attribute`

Dokładny opis działania oraz zwracane wartości opisuje dokumentacja znajdująca się na dołączonej płycie CD. Wyjaśnienie zasad programowania przy pomocy funkcji `scgl_edge_foreach_attribu` zostało przedstawione na stronie 43 pracy.

Ponieważ wartości atrybutów przetrzymywane są jako wskaźnik ogólny (`void*`), brak jest tu informacji na temat jej typu. Może być to problematyczne podczas pisania funkcji niszczącej atrybuty (uwalniającej dynamicznie przydzieloną im pamięć). Programista korzystając cały czas, z takich samych typów wartości atrybutów nie ma powodów do zmartwień. Chcąc jednak posiadać szereg atrybutów o różnych typach, można skorzystać z dwóch proponowanych, przez autora pracy, rozwiązań:

enum — wykorzystać typ wyliczeniowy `enum` do przechowywania informacji o typie wartości. Programista musiałby wówczas stworzyć własną strukturę, w której przechowywałby docelową wartość atrybutu krawędzi oraz jej typ.

sufiks — do klucza atrybutu można dołączyć ciąg znaków, który będzie definiował typ wartości atrybutu. Ciąg ten może być poprzedzany znakiem „:” (np. `lenght:d`, `lenght:s`) dzięki czemu łatwiej będzie parsować wartość klucza przed użyciem wartości atrybutu. Można wykorzystać do tego funkcję `strtok` znajdującą się w pliku nagłówkowym `string.h`.

Istnieje możliwość rozszerzenia biblioteki SCGL o atrybuty grafów oraz węzłów. Zrezygnowano jednak z implementacji tych funkcjonalności ze względu na chęć zużycia jak najmniejszej ilości pamięci. Przy projektowaniu biblioteki postanowiono, że atrybuty krawędzi są najistotniejsze i dlatego tylko dla tego modułu zostanie dopisana możliwość manipulacji nimi.

Funkcje `foreach`

Każdy z trzech modułów (graph, edge, vertex) posiada specjalną funkcję z sufiksem `_foreach`, jest to metoda służąca do wykonania funkcji użytkownika na kolejno wszystkich elementach (w przypadku krawędzi będą to atrybuty, w przypadku grafu krawędzie lub węzły). Może ona zostać wykorzystana do dowolnego celu, np. wypisania wszystkich atrybutów na ekran, lub przetworzenia ich wartości do kosztu/wagi krawędzi.

Funkcje użytkownika zdefiniowane są w odpowiednich plikach nagłówkowych, jednakże najczęściej przyjmują one następującą formę:

Listing 2.13. Przykładowa definicja funkcji użytkownika na potrzeby funkcji `foreach`

```
typedef void (*edge_foreach_function)(scgl_edge_t *edge, void **data);
```

Metody te nie zwracają żadnych zmiennych. Jako pierwszy argument przyjmują obiekt na którym wykonane zostaną działania, jako drugi dodatkowe dane, lub wskaźnik który przechowa referencję do rezultatów.

Przykładem użycia może być funkcja zrzucająca atrybut krawędzi do strumienia:

Listing 2.14. Przykładowa funkcja wykorzystywana przez metodę scgl_edge_dump

```
1 void edge_attr_dump(char *key, void *value, void **fp) {
2   if (fp != NULL && *fp != NULL)
3     fprintf((FILE*)*fp, "\t\t%s : %s \n", key, (char*)value);
4 }
```

Funkcja ta wywoływana będzie przez metodę `scgl_edge_dump` (zrzucanie krawędzi do strumienia) W tym przypadku mamy dwa argumenty określające obiekt na którym funkcja będzie operować, oraz podwójny wskaźnik na dodatkowe dane. Dane te będą wskaźnikiem na strumień (`FILE`), wiedząc to, możemy wykonać rzutowanie przedstawione w wierszu 3. Przed wykorzystaniem tego wskaźnika należy oczywiście wcześniej sprawdzić czy nie jest on zerowy (wiersz 2).

Funkcje typu `foreach`, mogą mieć wiele zastosowań, jednocześnie przyśpieszając pisanie oraz upraszczając kod użytkownika. Warto również wspomnieć, że funkcje te mogą zostać zastąpione przez wykorzystanie metod iteracyjnych `_get_at()`, które zwracają i-ty element listy oraz funkcji `_get_count`.

2.3.2. Kompilacja

Kompilacja zarówno biblioteki jak i testów jednostkowych oraz kodów użytkownika powinna odbywać się przy wykorzystaniu dostarczonego pliku *Makefile*. Jest to plik opisujący reguły postępowania dla programu automatyzującego proces kompilacji — *make*.

Plik Makefile, dla biblioteki SCGL znajduje się w jej głównym katalogu `scgl/`. Można wydzielić w nim trzy sekcje logiczne:

- część definiującą potrzebne zmienne, oraz pliki poddawane kompilacji,
- część decydującą o typie kosztu krawędzi (więcej na str. 28),
- część definiującą zależności oraz przebieg procesu kompilacji.

Dopisując nowy moduł do biblioteki SCGL powinniśmy dopisać jego pliki źródłowe (`*.c`) do zmiennej `SOURCES`. Makefile automatycznie wygeneruje reguły budowy plików obiektowych (`*.o`) bazując na nazwie pliku źródłowego. Pliki obiektowe linkowane (dołączane) są później do pliku biblioteki SCGL `libscgl.*`, który może zostać załączony do kodów źródłowych użytkownika.

Chcąc skompilować bibliotekę z użyciem domyślnego typu kosztu krawędzi (`unsigned int`) wystarczy wywołać:

```
make
```

Chcąc wybrać inny typ, powinno się zdefiniować wartość zmiennej `COST_TYPE` wg tabeli 1:

```
make COST_TYPE=d
```

Powyższe wywołanie skompiluje bibliotekę z użyciem kosztu krawędzi jako `double`.

```
make clean
```

Skasuje pliki cząstkowe (pliki obiektowe), wynikowe (libscgl.*) oraz te wyprodukowane przez DejaGNU. Warto wykonywać to polecenie, jeśli wprowadzamy jakieś zmiany w samej bibliotece SCGL.

Aby uruchomić procedurę testów jednostkowych, użytkownik powinien wywołać program `make` z celem `tests`:

```
make tests
```

Krok ten zbuduje bibliotekę SCGL (o ile nie została wcześniej zbudowana), zbuduje program odpowiedzialny za interfejs testowy a następnie uruchomi platformę DejaGNU.

Dodając własny cel (kod użytkownika) do pliku Makefile, użytkownik może chcieć skompilować go z biblioteką SCGL. Aby zrobić to poprawnie powinien skorzystać z poniższego przykładu:

```
my_app:
  @echo "Building my application"
  @$(CC) $(CFLAGS) $(MFLAGS) $(TOPDIR)my_dir/my_app.c -o $(TOPDIR)my_dir/my_app $(TOPDIR)
      lib/libscgl.a
```

W projekcie znajdują się również trzy dodatkowe pliki Makefile:

1. `scgl/doc/latex/` — plik generujący dokumentację doxygen do pliku `*.pdf`
2. `scgl/doc/thesis/` — plik generujący niniejszą pracę do pliku `*.pdf`
3. `scgl/perf_tests/` — plik kompilujący testy wydajności, wymaga zainstalowania w systemie bibliotek Boost::BGL oraz igraph.

3. Porównanie z istniejącymi rozwiązaniami

Jak wspomniano we wstępnie niniejszej pracy, teoria grafów jest jedną z najpotrzebniejszych dziedzin matematyki w informatyce. Ze względu na szeroką gamę zastosowań, powstało wiele rozwiązań implementujących w mniejszym, lub większym stopniu ową dziedzinę. Są to biblioteki napisane w niemal każdym języku programowania: C, C++, D, Python, Haskell, Matlab.

Jednym z głównych celów pracy, było stworzenie oprogramowania zużywającego jak najmniej zasobów komputera, zarówno pamięciowych jak i obliczeniowych. Aby potwierdzić spełnienie założeń należy porównać stworzoną bibliotekę, z innymi rozwiązaniami. Do porównań wybrane zostały dwie najpopularniejsze biblioteki grafów:

- Boost::BGL — część ogromnej biblioteki Boost, napisanej w języku C++
- igraph — oprogramowanie napisane w języku C

Boost::BGL Biblioteka BGL (ang. *Boost Graph Library*), została stworzona przez Jeremiego Sieka oraz grupę z uniwersytetu Notre Dame w USA. Wchodzi w skład bibliotek Boost, zawiera szablony reprezentujące grafy oraz zbiór kilkudziesięciu algorytmów grafowych[9]. Grafy są dostarczane w postaci generycznej, dzięki czemu z wierzchołkiem, krawędzią lub całym grafem można związać obiekt lub obiekty dowolnego typu. Szablony z biblioteki boost::graph wykorzystują kolekcje ze standardowej biblioteki szablonów (STL).

Graf w Boost::BGL może być reprezentowany listą sąsiedztwa, a także przez macierz sąsiedztwa albo w specjalnej postaci skompresowanej. W najprostszym przypadku, gdy krawędzie są przechowywane w wektorze, identyfikatorem wierzchołka jest liczba całkowita (indeks).

igraph *igraph* jest przestronną biblioteką grafów, wspierającą aplikacje pisane w językach takich jak: C, R, Python, czy Ruby. Zawiera funkcje implementujące klasyczne pro-

blemy teorii grafów, takie jak minimalne drzewa rozpinające, czy algorytmy najkrótszych ścieżek. Potrafi również generować kilkanaście rodzajów grafów, a także eksportować je do wielu formatów obsługiwanych przez oprogramowanie graficzne[14].

Jest najpopularniejszą biblioteką grafów napisaną w języku C.

3.1. Testy porównawcze

Porównania wcześniej wymienionych bibliotek z biblioteką SCGL dotyczyły wydajności obliczeniowej oraz pamięciowej, tych rozwiązań. Na potrzeby testów stworzono trzy scenariusze testowe:

1. Stopnia zużycia pamięci operacyjnej — dokładniej maksymalnego użycia sterty (ang. *heap*).
2. Szybkości tworzenia oraz usuwania obiektów.
3. Szybkości wykonywania algorytmu Dijkstry — obliczania najkrótszych ścieżek.

Każdy z nich przeprowadzony został w dwóch wariantach:

- z wykorzystaniem krawędzi skierowanych
- z wykorzystaniem krawędzi nieskierowanych

Pliki źródłowe testów, znajdują się w katalogu `perf_tests`. Przedrostek, w nazwie pliku, wskazuje na rodzaj testu, następna litera `d` na wariant „skierowany” (`u` na „nieskierowany”) a ostatni znak na wykorzystaną bibliotekę: `s` — SCGL; `b` — BGL; `i` — igraph.

Wszystkie testy zostały przeprowadzone w tych samych warunkach. Badania wykonano na jednordzeniowym procesorze Intel Celeron M420 taktowanym 1,6GHz oraz na pamięci RAM 400MHz (5 – 5 – 5 – 18 @ CL – RCD – RP – RAS). System pod którym przeprowadzono eksperymenty to GNU/Linux Arch z jądrem w wersji: 3.4.0-1. Programy zostały skompilowane przy użyciu:

- gcc 4.7.0 z następującymi flagami: -s -Os
- g++ 4.7.0 z następującymi flagami: -s -Os

Testy mające na celu porównanie użycia czasu procesora, powtarzane były w pętli tysiąckrotnie, a następnie uśredniane. Skrypt odpowiadający za powtarzanie testu przedstawiony jest na poniższym listingu:

Listing 3.1. Skrypt uśredniający czas wykonywania określonego określonego poleceniaa

```
#!/bin/bash
TIMEFORMAT="%U"
sum=0
for i in {1..1000}
do
  t=$( { time $1; } 2>&1 )
  sum=$( bc <<< "scale=10; $sum + $t" )
done
avrg=$( bc <<< "scale=3; $sum / 1000" )
echo "$avrg"
```

Jak już zostało wspomniane powtarza on wykonanie polecenia (przyjmowanego jako argument skryptu) 1000 razy, sumując czas użycia procesora, zmierzony programem *time* (wartości *user time*). Testy porównujące maksymalne użycie sterty, wykonane zostały przy pomocy programu *valgrind*[1].

Test mający na celu porównać poziom wykorzystania sterty przez wybrane biblioteki, polegał na stworzeniu jednego grafu, 1000 krawędzi oraz 1001 węzłów. Żaden ze stworzonych elementów nie przechowywał dodatkowych atrybutów.

Tablica 2. Wyniki pomiaru wykorzystania sterty

Nazwa biblioteki	Zużycie sterty w bajtach (ilość alokacji)	
	Graf skierowany	Graf nieskierowany
SCGL	76 048 (2002)	124 048 (3003)
BGL	44 753 (2012)	76 745 (3012)
igraph	112 160 (0021)	112 160 (0021)

Jak można zauważyć w powyższej tabeli, biblioteki BGL oraz SCGL wykorzystują więcej pamięci dla grafów nieskierowanych (wszystkie krawędzie są nieskierowane), niż dla grafów w pełni skierowanych. W przypadku SCGL wynika to z faktu, iż dostosowana ona jest do przetrzymywania krawędzi skierowanych. Obiekty nieskierowane tworzone są

[1] valgrind — jest narzędziem do debugowania pamięci, wykrywania wycieków pamięci oraz profilowania aplikacji.

poprzez dołączenie do krawędzi skierowanej jej odpowiednika z zamienionymi końcami (od/do). Aby ograniczyć zużycie pamięci, odpowiednik ten nie przechowuje informacji na temat atrybutów. Nie jest to wymagane, gdyż krawędź ta (odpowiednik), jest niejako zawieszony w powietrzu i nie jest dostępny z poziomu samego grafu.

Wyniki pomiarów, zamieszczone w tabeli 2, ukazują także, że zużycie pamięci dla dostarczonego przez autora pracy rozwiązania, są dużo mniejsze niż biblioteki igraph — w przypadku grafów skierowanych. Niestety nie udało się zmniejszyć zapotrzebowania, poniżej poziomu biblioteki BGL, mimo że osiągnięto porównywalną liczbę alokacji. Program `valgrind` pokazał w przypadku kodu stosującego oprogramowanie igraph, bardzo niską liczbę alokacji (21). Może to wynikać z faktu zastosowania w tym rozwiązaniu niestandardowego mechanizmu przydzielania pamięci.

Celem kolejnego z przeprowadzonych testów, było porównanie czasu procesora, jaki zostanie zużyty do tysiąckrotnej alokacji i uwolnienia obiektów z poprzedniego testu (1 graf, 1000 krawędzi, 1001 węzłów).

Tablica 3. Wyniki pomiaru wykorzystania czasu procesora podczas tworzenia/niszczenia obiektów

Nazwa biblioteki	Wykorzystanie czasu procesora w sek.	
	Graf skierowany	Graf nieskierowany
SCGL	0.384	0.560
BGL	0.563	0.740
igraph	0.295	0.295

Jak wynika z powyższych rezultatów, niska ilość alokacji biblioteki igraph przekłada się na szybkość wykonywania tych operacji. Czasy osiągane przez to oprogramowanie są niemal dwukrotnie niższe, niż w przypadku biblioteki BGL. Ponownie w przypadku SCGL oraz BGL, wyniki dla grafów nieskierowanych są gorsze niż dla skierowanych. Biblioteka SCGL osiągnęła niższe czasy tworzenia i czyszczenia pamięci niż jej odpowiednik napisany w C++.

Ostatni test polegał na wytyczeniu najkrótszych ścieżek z pierwszego węzła w grafie (indeks zerowy), do wszystkich pozostałych, przy zastosowaniu algorytmu Dijkstry (więcej na str. 33). Podobnie jak poprzednie testy, ten bazował na dwóch grafach: skierowanym oraz nieskierowanym, oba o nieujemnych kosztach poszczególnych krawędzi, reprezento-

wane są przez rysunki 11, 10.

Z powodu niewielkich rozmiarów badanych grafów, operacja wyznaczania ścieżek została powtórzona 10000 razy.

Tablica 4. Wyniki pomiaru wykorzystania czasu procesora podczas wykonywania algorytmu Dijkstry

Nazwa biblioteki	Wykorzystanie czasu procesora w sek.	
	Graf skierowany	Graf nieskierowany
SCGL	0.036	0.052
BGL	0.036	0.049
igraph	0.477	0.800

Jak można zauważyć w powyższej tabeli, wyniki biblioteki igraph znacząco odstają od rozwiązań SCGL oraz BGL. Powód tego faktu pozostaje nieznany, jednakże może to wynikać ze sposobu przechowywania informacji (na temat grafu) w pamięci.

Stworzona na potrzeby celów pracy, biblioteka SCGL, osiąga porównywalne czasy do biblioteki BGL. Warto zauważyć, że SCGL korzysta z mniej wydajnych list powiązanych, co za tym idzie, musi wykonywać dodatkowe czynności w czasie wybierania kolejnych węzłów. Z tego powodu, w przypadku dużych grafów, implementacja ta będzie mniej wydajna niż BGL. Różnica pomiędzy czasami dla grafu skierowanego i nieskierowanego, dla biblioteki SCGL, wynika z dodatkowej ilości obiektów (dodatkowe krawędzie symulują „bezkierunkowości"), które musi przeszukać algorytm.

Dodatkowym atutem biblioteki SCGL jest jej prostota. Z tego powodu postanowiono przeprowadzić również porównanie rozmiarów plików wynikowych z linkowaniem statycznym (ang. *static linking*).

Konsolidacja statyczna, polega na umieszczeniu plików biblioteki (pliki z rozszerzeniem `*.a`) wewnątrz aplikacji użytkownika. Dzięki temu zabiegowi, aplikacja ta uniezależniona jest od współdzielonych plików (znajdujących się najczęściej w `/usr/lib/`), które mogą ulec zmianie lub zniszczeniu — co czyni ją w pełni przenośną (pomijając kwestie architektury komputera). Taki sposób kompilacji wykorzystuje się również często w rozwiązaniach opartych o systemy wbudowane (ang. *embedded*), gdzie środowisko systemowe jest ograniczone, a aplikacja użytkownika powinna być przenośna.

Testom zostały poddane aplikacje wykorzystane przy, wcześniej opisanych, testach wydajnościowych.

- `mem_size` — test wykorzystania pamięci
- `mem_speed` — test szybkości tworzenia/usuwania obiektów
- `dijkstra` — test szybkości działania algorytmu Dijkstry

Tablica 5. Wyniki pomiaru rozmiarów plików wykonywalnych z zastosowaniem różnych typów konsolidacji

Nazwa biblioteki	Nazwa aplikacji	Wynik konsolidacji statycznej w KiB	Wynik konsolidacji dynamicznej w KiB
SCGL	`mem_size`	736	3.8
	`mem_speed`	736	3.8
	`dijkstra`	742	4.8
BGL	`mem_size`	1638	9.2
	`mem_speed`	1638	9.2
	`dijkstra`	1740	21.0
igraph	`mem_size`	884	3.9
	`mem_speed`	884	3.9
	`dijkstra`	1126	5.1

Dla porównania zamieszczono również rozmiary aplikacji po linkowaniu dynamicznym. Jak można zauważyć na powyższej tabeli, rozwiązanie proponowane przez autora pracy, charakteryzuje się najmniejszymi rozmiarami plików. Cecha ta może być ogromną zaletą podczas tworzenia aplikacji na wspominane systemy wbudowane. Widać również, że fakt wykorzystania języka C++ do stworzenia biblioteki BGL znacząco wpływa na jej rozmiary (podobną zależność autor pracy zaobserwował we wszystkich, nawet najprostszych programach napisanych w C++).

Jako ciekawostkę można dodać, że kody źródłowe oparte na bibliotece BGL, jak i wszystkie rozwiązania oparte na języku C++, mają przeciętnie 3—4 razy dłuższe czasy kompilacji, niż ich odpowiedniki w języku C (zarówno dla biblioteki igraph, jak i SCGL).

Podsumowanie

W ramach niniejszej pracy dyplomowej, według opinii jej autora, zrealizowano wszystkie postawione w niej cele i wymagania.

Zaimplementowano podstawową bibliotekę grafów w języku C — *Simple C Graph Library: SCGL*. Biblioteka ta podzielona została na moduły zajmujące się obsługą poszczególnych elementów teorii grafów. Zaprojektowano oraz wdrożono jednolity interfejs programisty (API), który pozwala na dodanie obsługi grafów w aplikacjach innego użytkownika. Dodatkowo do biblioteki została załączona pełna dokumentacja owego API, stworzona w języku angielskim. SCGL obsługuje również jeden z najważniejszych algorytmów teorii grafów — algorytm wyszukiwania najkrótszych ścieżek Dijkstry.

Praca ta nie stanowi nowego podejścia do problemu, istnieje bowiem ogromna ilość bibliotek grafowych, w tym również te napisane w języku C. Jednakże jest to jedna z bardzo nielicznych (o ile nie jedyna) prosta implementacja podstawowych elementów teorii grafów. Dzięki temu może ona stanowić przykład pozwalający zrozumieć zasady działania algorytmów grafowych.

Zaimplementowana biblioteka potrafi tworzyć, manipulować oraz usuwać zarówno grafy jak i ich elementy składowe (wierzchołki oraz krawędzie) a także wyznaczać najkrótsze ścieżki przy pomocy algorytmu Dijkstry. Pozwala ona również na przechowywanie dowolnej liczby, dowolnych typów atrybutów krawędzi, a także dostosowanie typu atrybutu kosztu krawędzi do potrzeb użytkownika (typ kompilowany statycznie). Cechuje się przede wszystkim prostotą oraz niezbyt dużą ilością kodów źródłowych. Dzięki czemu pozostaje czytelna, co znacznie ułatwi przyszły rozwój tego oprogramowania. Dodatkowo załączone do pracy, na płycie CD, repozytorium programu *git*, pozwoli na poznanie historii tworzenia biblioteki.

Prostota zaproponowanego rozwiązania przekłada się również na jego wydajność. Wykorzystując grafy skierowane SCGL zużywa mniej pamięci RAM niż biblioteka igraph. Jest ona również dużo szybsza w przypadku tworzenia oraz usuwania obiektów od rozwiąza-

nia proponowanego przez twórców Boost::Graph. Dodatkowo wykazała się podobnym (do BGL) czasem wykonywania algorytmu Dijkstry dla niewielkich grafów, a dużo mniejszym do igraph. Biblioteka SCGL cechuje się również najmniejszymi, spośród badanych rozwiązań, rozmiarami plików wykonywalnych (zarówno w wyniku konsolidacji dynamicznej jak i statycznej). Dzięki temu może być stosowana w rozwiązaniach wbudowanych (embedded) gdzie występuje potrzeba minimalizacji zużywanej pamięci masowej FLASH/EEPROM.

Pomimo faktu, że biblioteka ta została napisana z myślą o systemach operacyjnych rodziny Unix/Linux, z całą pewnością może być ona przeniesiona na inne systemy operacyjne (np. firmy Microsoft). Pewność ta wynika z braku zależności biblioteki od jakichkolwiek specyficznych mechanizmów systemów Unix/Linux.

Stworzoną do celów pracy bibliotekę, można by w przyszłości rozbudować realizując zaproponowane przez autora pracy pomysły:

- Dodanie kolejnych algorytmów teorii grafów, np. algorytm Bellmana-Forda, lub Prima.
- Przyśpieszenie działania algorytmu Dijkstry poprzez zastosowanie tabel asocjacyjnych.
- Dostosowanie ilości zmiennych „koszt", wewnątrz krawędzi, do potrzeb użytkownika.
- Ukrycie przed użytkownikiem biblioteki zawartości wszystkich struktur, wymuszając korzystanie jedynie z dostarczonego API.
- Dodanie mechanizmu serializacji grafów, dzięki któremu biblioteka mogłaby być wykorzystana w aplikacjach korzystających z MPI[2].

Można by również pokusić się o stosowanie statycznie kompilowanych zmiennych w miejsce aktualnych atrybutów dynamicznych. Pozwoliłoby to na jeszcze dokładniejsze dostosowanie biblioteki do aktulanych potrzeb użytkownika, jak również przyśpieszyłoby jej działanie. Wiązałoby się to jednak ze wzrostem komplikacji problemu zależności, jednakże dużym ułatwieniem byłoby zastosowanie oprogramowania *kconfig*.

[2]MPI (ang. *Message Passing Interface*, protokół komunikacyjny będący standardem przesyłania komunikatów pomiędzy procesami programów równoległych działających na jednym lub więcej komputerach.

W dalszym rozwoju biblioteki może pomóc fakt istnienia szeregu testów jednostkowych, napisanych przy wykorzystaniu platformy DejaGNU. Pozwalają one zweryfikować poprawność działania istniejących już modułów/jednostek biblioteki SCGL, przez co programista może mieć pewność, że jego zmiany nie wprowadziły dodatkowych błędów.

Autor pracy zebrał oraz przyswoił wiedzę na temat podstawowych aspektów teorii grafów, oraz działania algorytmów wyszukiwania najkrótszych ścieżek. Pogłębił również wiedzę z tematyki programowania w języku C. Zapoznał się dokładniej z technologiami wykorzystywanymi w jądrze systemów Linux, oraz platformą testową DejaGNU. Posiadł umiejętności z zakresu tworzenia bibliotek zarówno statycznych jak i dynamicznych, oraz testów jednostkowych. Poszerzył także wiedzę o dobrych praktykach programistycznych.

Podsumowując można stwierdzić, że zrealizowana w ramach niniejszej pracy implementacja podstawowej biblioteki grafów, może być skutecznie wykorzystywana do modelowania zjawisk świata rzeczywistego (np. sieci komputerowych). Może być również niezwykle użyteczna w procesie analizy zasad działania algorytmów teorii grafów, a także do przybliżenia technik programistycznych umożliwiających tworzenie wydajnego oprogramowania.

Summary

The work describes the desing and the development of a basic graph library in C language. Mechanic of the designed library and the user API are described in detail and UML diagrams are being shown.

Within the work author created an basic graph library and implemented Dijkstra shortest path algorithm. Author also designed and implemented unit tests for core functionality.

The created software fully benefits from the advantages of C language. It is generall smaller and faster than other popular graph libraries (Boost::Graph, igprah). The API is well described by the generated documentation, which is in English.

The created library, it's source code and documentation are attached on the compact disc.

Bibliografia

[1] Ross K.A., Wright C., *Matematyka dyskretna*, PWN, 2006

[2] West D.B, *Introduction to Graph Theory*, Prentice Hall, 1999

[3] Cormen, T.H., Leiserson, C.E., *Introduction to Algorithms*, MIT Press, 2009

[4] Moy J.T, *OSPF: Anatomy of an Internet Routing Protocol*, Addison-Wesley, 1998

[5] Raymond S.E., *The Art of UNIX Programming*, Addison-Wesley Professional, 2003

[6] Kernighan B.W., Ritchie D.M., *Język ANSI C. Programowanie.* Helion, 2010

[7] Banachowski L., *Algorytmy i struktury danych*, WNT 2011

[8] Chen M., Roche D.L., *Priority Queues and Dijkstra's Algorithm*, Fujitsu, 2007

[9] Boost Graph Library: Dijkstra's Shortest Paths, `http://www.boost.org/doc/libs/1_49_0/libs/graph/doc/dijkstra_shortest_paths.html`, stan na dzień 15.03.2012

[10] Linux Kernel Linked List Explained, `http://isis.poly.edu/kulesh/stuff/src/klist/`, stan na dzień 25.02.2012

[11] FAQ/LinkedLists, `http://kernelnewbies.org/FAQ/LinkedLists`, stan na dzień 26.02.2012

[12] Doar M.B., *Practical Development Environments*, O'Reilly Media, 2005

[13] Libes D., *Exploring Expect*, O'Reilly Media, 2010

[14] The igraph library for complex network research, `http://igraph.sourceforge.net/documentation.html`, stan na dzień 01.12.2011

Dodatek A. SCGL Tutorial

This appendix was written to help getting started with the SCGL. It contains descriptions of simple examples of using that library.

In all the following examples user have to attach appropriate include files (eg `stdlib.h`). To use the SCGL library is needed only to include additional `scgl.h`.

Listing 3.2. Creating a graph, vertexes and edges

```
1  scgl_graph_t *g1;
2  scgl_vertex_t *v1, *v2;
3  scgl_edge_t *e1;
4  char *buf;
5
6  buf = (char*) malloc(4);
7  sprintf(buf, "V-1");
8  v1 = scgl_vertex_create(buf, NULL, 0, NULL, 0);
9  v2 = scgl_vertex_create(NULL, NULL, 0, NULL, 0);
10 e1 = scgl_edge_create(v1, v2, 0, 123, NULL, 0);
11 g1 = scgl_graph_create(NULL, NULL, 0, NULL, 0);
12
13 scgl_graph_add_vertex(g1, v1);
14 scgl_graph_add_vertex(g1, v2);
15 scgl_graph_add_edge(g1, e1);
16
17 scgl_graph_destroy(&g1);
```

The above listing shows how to create basic SCGL's objects: graph, vertex, edge. It use pointers because of author's habits. It is important to use typedefined variable types eg `scgl_graph_t`, because in future access to struct's members will be limited, through that mechanism.

It isn't required to create objects (vertexes, graphs) with id, but vertex `v1` was created in that way. Lines 6–7 shows how to fill allocated memory with proper string, and next how to pass that buffer into vertex's constructor.

In this example also created directed edge (agr no. 3), with travel cost equals 123. That edge connects vertex `v1` with `v2`. In that case, connection were created through

edge's constructor, but it can be replaced by the use of function `scgl_edge_add_vertex` or `scgl_vertex_add_edge`.

Because code doesn't store vertexes and edges in arrays, it can't pass that information to graph's constructor (arg no. 2, 3, 4 and 5), so it forces the use of other functions (lines 13–15) to add objects to graph.

A good practice is to release the previously allocated memory (by object creation). In the last line of listing, there is a call of graph's destroy function, which will destroy all objects (also vertex id) linked with graph instance passed as argument.

Listing 3.3. Managing edge's attributes

```
1  void edge_attr_free(char *key, void *value, void **data) {
2    free(value);
3  }
4
5  scgl_edge_t *e1;
6  char *buf1, buf2;
7
8  buf1 = (char*) malloc(5);
9  buf2 = (char*) malloc(5);
10 sprintf(buf1, "key1");
11 sprintf(buf2, "val1");
12
13 e1 = scgl_edge_create(NULL, NULL, 0, 1, NULL, 0);
14 scgl_edge_add_attribute(e1, buf1, buf2);
15 printf("%s \n", (char*)scgl_edge_get_attribute_value(e1, "key1"));
16
17 scgl_edge_destroy(&e1, edge_attr_free);
```

The above listing is an example of creating, getting and destroying edge's attributes. As before, buffers are filled by `sprintf` function, and passed to function which will add new attribute to edge. Attribute will have key equals `buf1` and value equals `buf2`. To get attribute value it is necessary to pass key to function `scgl_edge_get_attribute_value`. It will look for that key among all edge's attributes.

Attribute's value can be everything (`void*`), so it is important to write a user function which will destroy memory occupied by that pointer. Lines 1–3 show example of that function. It just only free value because in that case value is simple string. In the last line of listing, that function is passed into edge's destroy method.

Last example in this chapter shows complete code setting out shortest path from vertex 0, using Dijkstra's algorithm. Figure 1 shows graph which will be created in the following

code:

Listing 3.4. Use of Dijkstra's algorithm example

```
scgl_graph_t *g1;
scgl_vertex_t **v;
scgl_edge_t **e;

unsigned int *p;
cost_type_t *d;
unsigned int i, n = 4, m = 5;

v = (scgl_vertex_t**) malloc(sizeof(scgl_vertex_t*) * n);
for(i=0; i<n; ++i)
  v[i] = scgl_vertex_create(NULL, NULL, 0, NULL, 0);

e = (scgl_edge_t**) malloc(sizeof(scgl_edge_t*) * m);
for(i=0; i<m; ++i)
  e[i] = scgl_edge_create(NULL, NULL, undirected, 0, NULL, 0);

g1 = scgl_graph_create(NULL, v, n, e, m);

scgl_edge_add_vertex(e[0], v[0], 0);
scgl_edge_add_vertex(e[0], v[1], 1);
scgl_edge_add_vertex(e[1], v[1], 0);
scgl_edge_add_vertex(e[1], v[2], 1);
scgl_edge_add_vertex(e[2], v[2], 0);
scgl_edge_add_vertex(e[2], v[3], 1);
scgl_edge_add_vertex(e[3], v[3], 0);
scgl_edge_add_vertex(e[3], v[1], 1);
scgl_edge_add_vertex(e[4], v[0], 0);
scgl_edge_add_vertex(e[4], v[2], 1);
scgl_edge_set_cost(e[0], 1);
scgl_edge_set_cost(e[1], 2);
scgl_edge_set_cost(e[2], 3);
scgl_edge_set_cost(e[3], 4);
scgl_edge_set_cost(e[4], 5);

p = (unsigned int*) malloc(sizeof(unsigned int) * n);
d = (cost_type_t*) malloc(sizeof(cost_type_t) * n);

scgl_dijkstra(g1, v[0], p, d);
for(i=0; i<n; ++i)
  printf("%d, ", p[i]);
for(i=0; i<n; ++i)
  printf(cost_fmt ", ", d[i]);
printf("\n");

scgl_graph_destroy(&g1, edge_attr_free);
```

```
46 free(p);
47 free(d);
48 free(v);
49 free(e);
```

Table `d` is an array of `cost_type`, it will contain distances to every vertex in graph, from vertex 0. `Cost_type` is an typedef replaced by compilator with user chosen type. Table `p` is an array of predecessors, by reading it user can restore shortest path from vertex 0 to desired point.

This set of examples is only to help user get started with the SCGL library. The real source of information is the documentation written in English, located on the CD, attached to this thesis.

Dodatek B. Oświadczenie

.

Imię i nazwisko: *Patryk Kwiatkowski* Częstochowa, dn. *7 czerwca 2012*

Nr albumu: *101510*

Kierunek: *Informatyka*

Wydział: *Wydział Inżynierii Mechanicznej i Informatyki*

Politechnika Częstochowska

Szanowny Pan (i) Dziekan

Oświadczenie

Pod rygorem odpowiedzialności karnej oświadczam, że złożona przez mnie praca dyplomowa pt. *„Implementacja podstawowej biblioteki grafów w języku C”* jest moim samodzielnym opracowaniem.

Jednocześnie oświadczam, że praca w całości lub we fragmentach nie została dotychczas przedłożona w żadnej szkole.

Niezależnie od art.239 Prawo o szkolnictwie wyższym wyrażam zgodę na nieodpłatne wykorzystanie przez Politechnikę Częstochowską całości lub fragmentów w/w pracy w publikacjach Politechniki Częstochowskiej.

.....

podpis

Dodatek C. Opis zawartości płyty CD

Listing 3.5. Struktura plików zamieszczonych na płycie CD

```
.
|-- .git/
|-- doc/
|   |-- latex/
|   \-- thesis/
|       |-- gfx/
|       |-- thesis.pdf
|       |-- thesis.tex
|       \-- thesis.txt
|-- include/
|-- perf_tests/
|-- src/
|-- unit_tests/
|   \-- scgl.test/
\-- LICENSE
```

.git/ katalog zawierający repozytorium programu *git*

doc/latex dokumentacja wygenerowana przez program *Doxygen*

doc/thesis/gfx/ katalog zawierający grafiki wykorzystane w niniejszej pracy magisterskiej

doc/thesis/thesis.pdf treść pracy zapisana w formacie Adobe Portable Document

doc/thesis/thesis.tex treść pracy zapisana w języku LaTeX

doc/thesis/thesis.txt treść pracy zapisana w formacie tekstowym

include/ zawiera pliki nagłówkowe opisanej w pracy biblioteki SCGL

perf_tests/ zawiera pliki źródłowe testów wydajnościowych

src/ zawiera pliki kodu źródłowego opisanej w pracy biblioteki SCGL

unit_tests/scgl.test/ zawiera skrypt testów jednostkowych platformy DejaGNU oraz plik kod źródłowy interfejsu tych testów

LICENSE treść licencji na jakiej został udostępniony kod biblioteki SCGL

Spis rysunków

Spis tablic

Spis listingów