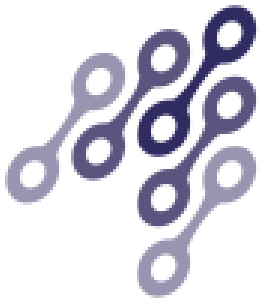
POLITECHNIKA CZĘSTOCHOWSKA

WYDZIAŁ INŻYNIERII MECHANICZNEJ I INFORMATYKI



PRACA INŻYNIERSKA

Interfejs biblioteki grafowej dla języka C++

Graph library interface for C++ language

Witold Krypczyk

Nr albumu: 102835

Kierunek: Informatyka

Promotor pracy: dr inż. Ireneusz Szcześniak

Praca przyjęta dnia:

Podpis promotora:

Częstochowa, 2012

Spis treści

Streszczenie

Użycie bibliotek języka C w oprogramowaniu tworzonym w języku C++ jest proste, jednak biblioteki takie nie umożliwiają wykorzystania wielu możliwości tego języka. Niniejsza praca omawia projekt, implementację oraz testowanie biblioteki igraphxx, dostarczającej interfejs języka C++, dla biblioteki języka C igraph, pozwalającej na tworzenie i manipulowanie grafami.

Pierwsza część pracy zawiera krótki opis biblioteki igraph, używanej przez nią struktury danych reprezentującej graf oraz omawia wybraną część jej interfejsu. Następnie omówione zostały wybrane cechy języka C++, takie jak przestrzenie nazw, funkcje składowe, automatyczna inicjacja, wyjątki, itp.

Dalej omówiono proces budowania zależności zaimplementowanej biblioteki oraz testów. W ich skład wchodzą biblioteki Igraph, Boost, Loki, Igraphpp.

Biblioteka igraphxx została zaimplementowana w postaci samych plików nagłówkowych. Zawiera interfejs ułatwiający używanie struktur danych biblioteki igraph, takich jak graf, wektor, czy iteratory, system umożliwiający użycie obiektów języka C++ jako atrybutów wierzchołków i krawędzi oraz funkcję sygnalizującą błędy biblioteki igraph za pomocą wyjątków języka C++.

W ramach pracy zaimplementowano także testy jednostkowe dla biblioteki z użyciem platformy Boost Test, porównano wydajność z interfejsem dla języka C oraz interfejsem dla języka C++ w formie biblioteki igraphpp oraz stworzono dokumentację biblioteki igraphxx z wykorzystaniem programu Doxygen.

Na końcu pracy umieszczono przykłady użycia stworzonej biblioteki.

Abstract

Using C language libraries in software created in C++ language is easy, but these libraries don't provide the ways to of using many feature possibilities of this programming language. This work is about the design, the implementation and the testing of software library called igraphxx, which provides C++ language interface for igraph. Igraph is C language library for creating and manipulating graphs.

The first part of the work describes igraph library. It contains a brief description of the library, the data structure used by it for representing the graphs, and discusses the selected part of its interface. Then, there are discussed some of the features of C++ language, such as namespaces, member functions, automatic initiation, exceptions, etc.

Next, the process of building dependencies of implemented library and tests is discussed. It includes Igraph, Boost, Loki and Igraphpp. Igraphxx library was implemented as a header-only library. It contains the interface which facilitates the usage of igraph data structures, such as graph, vector, or iterators, the system enables to use C++ objects as the attributes of vertices and edges, and the function which can report igraph library errors by means of C++ exceptions.

This work also contains the implementation of unit tests for the library with the usage of Boost Test Framework, it also includes the comparison of performance with C language interface, and with the interfaces for C++ called igraphpp, there was also created the documentation for igraphxx library using the Doxygen program.

At the end of this work there are examples of the usage of the created library.

1. Wstęp

1.1. Cel i zakres pracy

Celem niniejszej pracy jest zaprojektowanie i implementacja interfejsu stworzonej w języku C biblioteki igraph dla języka C++, jego przetestowanie oraz porównanie wydajności z interfejsem dla języka C oraz istniejącym, nieukończonym interfejsem dla języka C++ zaimplementowanym w postaci biblioteki igraphpp.

W ramach niniejszej pracy stworzona została biblioteka igraphxx, dostarczająca interfejs języka C++ dla biblioteki igraph, składający się z trzech niezależnych części. Pierwsza z nich pozwala na wykonywanie podstawowych operacji na reprezentującej grafy strukturze biblioteki igraph, umożliwiając tworzenie, modyfikowanie oraz sprawdzanie zawartości grafów. Druga pozwala na użycie dowolnych obiektów języka C++, jako atrybutów wierzchołków oraz krawędzi. Dzięki części trzeciej możliwe jest użycie wbudowanego w język C++ mechanizmu wyjątków, do sygnalizowania błędów biblioteki igraph.

Oprócz samej biblioteki, stworzono także dokumentację dla niej, wykorzystując przy tym narzędzie Doxygen.

W ramach pracy zaimplementowano również testy jednostkowe dla stworzonej biblioteki, wykorzystując przy tym platformę Boost Test. Nie udało się jednak uzyskać zakładanego pokrycia kodu testami.

Ostatnim krokiem było porównanie wydajności stworzonej biblioteki z interfejsem języka C oraz dostarczającą innego interfejsu języka C++ biblioteką igraphpp.

1.2. Struktura pracy

Praca składa się z dziewięciu rozdziałów. Pierwszy z nich opisuje cel pracy, osiągnięty stopień jego realizacji oraz strukturę pracy. Drugi rozdział zawiera omówienie biblioteki igraph. W rozdziale trzecim omówiono wybrane elementy języka C++.

Rozdział czwarty opisuje sposób budowy zależności stworzonej biblioteki, testów jednostkowych oraz testów wydajności. W piątym rozdziale zawarto krótkie omówienie interfejsu stworzonej biblioteki. Szósty z rozdziałów zawiera informacje o testach jednostkowych zaimplementowanych dla stworzonej biblioteki. W rozdziale siódmym znalazło się porównanie wydajności stworzonej biblioteki, z interfejsem biblioteki igraph dla języka C oraz innym jej interfejsem dla języka C++, zaimplementowanym w bibliotece igraphpp. Rozdział ósmy stanowi omówienie systemu doxygen, użytego do wygenerowania dokumentacji biblioteki igraphxx. Ostatni z rozdziałów zawiera przykłady użycia stworzonej biblioteki.

2. Biblioteka igraphg

2.1. O bibliotece

Igraph to napisana w języku C biblioteka, przeznaczona do operowania na grafach skierowanych oraz nieskierowanych, posiadająca także interfejsy dla języków GNU R, Python oraz Ruby.

Biblioteka ta pozwala na operowanie na grafach zawierających miliony wierzchołków oraz krawędzi, przechowywanie atrybutów wierzchołków i krawędzi, użycie wielu zaimplementowanych w niej algorytmów grafowych oraz odczyt i zapis grafów w plikach za pomocą kilku obsługiwanych formatów.

Igraph jest wolnym oprogramowaniem rozpowszechnianym na zasadach Powszechnej Licencji Publicznej GNU[4].

2.2. Struktura danych reprezentująca graf

W bibliotece igraph każdy wierzchołek oraz krawędź grafu posiadają etykiety będące kolejnymi liczbami całkowitymi rozpoczynając od zera, do odpowiednio liczby wierzchołków lub krawędzi minus jeden. Graf jest wielozbiorem krawędzi będących odpowiednio uporządkowanymi w przypadku grafów skierowanych oraz nieuporządkowanymi w przypadku grafów nieskierowanych parami etykiet wierzchołków. Użyta struktura pozwala przechowywać pseudografy, jednak większość funkcji biblioteki nie obsługuje wielu równoległych krawędzi. Biblioteka zawiera jedną strukturę języka C *igraph_t* reprezentującą zarówno grafy skierowane, jak i nieskierowane[4].

2.3. Operacje na zbiorach wierzchołków oraz krawędzi

Biblioteka igraph zawiera niezależne od grafu selektory, reprezentujące koncepcje grupy wierzchołków lub krawędzi. W bibliotece zaimplementowane są m.in. następujące koncep-

cje grup wierzchołków:

- Wszystkie wierzchołki w grafie.
- Sąsiedzi wskazanego wierzchołka. W przypadku grafów skierowanych, istnieje możliwość wybrania rodzaju sąsiedztwa spośród wierzchołków, dla których istnieje połączenie odpowiednio od lub do wskazanego wierzchołka oraz sumy obu wymienionych zbiorów.
- Wierzchołki nie sąsiadujące ze wskazanym wierzchołkiem. Istnieje możliwość wyboru rodzaju sąsiedztwa jak wyżej.
- Pusty zbiór wierzchołków.
- Jeden wskazany wierzchołek.
- Zbiór wierzchołków o podanych etykietach.
- Wierzchołki o etykietach będących kolejnymi liczbami całkowitymi.

Do koncepcji grup krawędzi należą m.in.:

- Wszystkie krawędzie w grafie.
- Krawędzie w sąsiedztwie wskazanego wierzchołka. W przypadku grafów skierowanych istnieje możliwość wyboru pomiędzy krawędziami do wskazanego wierzchołka, od wskazanego wierzchołka lub sumą wymienionych zbiorów.
- Pusty zbiór krawędzi.
- Jedna wskazana krawędź.
- Zbiór krawędzi o podanych etykietach.
- Krawędzie o etykietach będących kolejnymi liczbami całkowitymi.
- Krawędzie pomiędzy wierzchołkami o podanych etykietach.

Selektory używane są podczas operacji wymagających wybrania zbioru wierzchołków lub krawędzi, takich jak iteracja po ich etykietach, zliczanie, czy usuwanie oraz w wchodzących w skład biblioteki funkcji implementujących algorytmy grafowe, wymagających podania zbioru wierzchołków lub krawędzi[4].

2.4. Biblioteka struktur danych

Biblioteka igraph zawiera zbiór uniwersalnych struktur danych, takich jak wektor, macierz, stos, czy kolejka. Struktury te mogą zawierać wartości kilku różnych typów, zazwyczaj zmiennoprzecinkowego, całkowitego i logicznego. W bibliotece użyto do tego celu preprocesora, ponieważ język C nie wspiera programowania generycznego[4].

2.5. Obsługa błędów

W skład biblioteki igraph wchodzi własny system obsługi błędów. Kiedy wystąpi błąd wywoływana jest wskazana przez użytkownika funkcja, przyjmująca jako argumenty kod błędu, łańcuch znaków z tekstowym opisem powodu jego wystąpienia, nazwę pliku oraz numer linii, w której wystąpił błąd. Domyślnie ustawiona jest funkcja wypisująca wiadomość o błędzie oraz kończąca działanie programu. Wraz z biblioteką dostarczone są jeszcze funkcje odpowiednio ignorująca błąd kontynuując działanie programu oraz wypisująca informacje o błędzie i kontynuująca działanie programu. W przypadku kontynuacji wykonywania programu można wykryć wystąpienie błędu dzięki zwracaniu jego kodu przez większość funkcji wchodzących w skład interfejsu biblioteki. Poza wymienionymi funkcjami biblioteka ta dostarcza również możliwość stworzenia własnej funkcji obsługi błędów. W tym celu dostarczone zostało makro zwalniające tymczasowo przydzieloną pamięć. System obsługi błędów zastosowany w bibliotece nie nadaje się do stosowania w aplikacjach wielowątkowych ze względu na użycie globalnego stosu do przechowywania wskaźników na tymczasowo przydzielane obiekty[4].

2.6. Atrybuty grafów, wierzchołków oraz krawędzi

Biblioteka posiada interfejs pozwalający na użycie własnego systemu systemu przechowywania atrybutów przypisanych do grafów, wierzchołków oraz krawędzi o nazwie reprezentowanej w formie łańcucha znaków. System taki musi obsługiwać operacje takie jak modyfikacja zbiorów wierzchołków i krawędzi, tworzenie, kopiowanie i niszczenie grafu oraz opcjonalnie pobieranie i przypisywanie wartości atrybutu lub zbioru atrybutów w formie łańcucha znaków oraz liczby zmiennoprzecinkowej. Wraz z biblioteką dostarczony jest eksperymentalny system przechowywania atrybutów dla języka C pozwalający na

przechowywanie łańcuchów znaków oraz liczb zmiennoprzecinkowych jako atrybutów. Dedykowane systemy przechowywania atrybutów dostarczone są również wraz z interfejsami dla języków GNU R oraz Python.

W przypadku algorytmów grafowych wymagających atrybutów, takich jak wagi wierzchołków lub krawędzi, implementujące je funkcje przyjmują jako argument zawierający wektor[4].

3. Język C++ a język C

3.1. O języku C++

C++ to ustandaryzowany przez ANSI i ISO język programowania ogólnego przeznaczenia, oparty o język C, wzbogacony m.in. o dodatkowe typy danych, klasy, szablony, wyjątki, przestrzenie nazw, przeciążanie nazw funkcji i operatorów, menadżer dynamicznej pamięci oraz rozszerzoną bibliotekę standardową[5].

Jego początki sięgają 1979 roku, kiedy to Bjarne Stroustrup stworzył język nazywany C z klasami, łączący efektywność i elastyczność programowania systemowego w języku C z wspomagającymi organizację kodu klasami wzorowanymi na klasach języka Simula[10].

3.2. Łączenie kodu w obu językach

Język C++, poza drobnymi wyjątkami jak silniejsza kontrola typów, jest kompatybilny z językiem C na poziomie kodu źródłowego, dzięki czemu kod napisany w języku C może być skompilowany kompilatorem języka C++ i co za tym idzie użyty przez inny kod napisany w języku C++. Rozwiązanie to może być najlepsze nawet jeśli kod napisany w języku C wymaga zmian, ponieważ dzięki kompilacji kompilatorem języka C++ zyskujemy, ułatwiającą wykrycie błędów, silniejszą kontrolę typów oraz dodatkowe możliwości optymalizacji, których nie ma kompilator języka C. Rozwiązanie to nie nadaje się jednak do łączenia kodu w języku C++ ze skompilowanym już kodem języka C, jak np. binarne wersje bibliotek[7].

C++ został jednak zaprojektowany tak, aby jego linkowanie z fragmentami programów stworzonych m.in. w języku C było łatwe i efektywne[10]. Każda nazwa funkcji, nazwa zmiennej oraz typ funkcji posiadają język linkowania. Domyślnie jest to C++, jednak możliwe jest wyspecyfikowanie linkowania z językiem wspieranym przez daną implementację. Według standardu ISO każda implementacja powinna pozwalać na linkowanie z

funkcjami napisanymi w języku C. Aby było to możliwe konieczne jest odpowiednie wyspecyfikowanie sposobu linkowania, konkretnie poprzedzenie jej poprzez extern "C"[5]. Przykład:

```
extern ''C'' void funkcja(void);
```

Możliwe jest także wyspecyfikowanie dla całego bloku:

```
extern ''C''
{
  void funkcja1(void);
  void funkcja2(void);
}
```

Biblioteki języka C używają często makra preprocesora o nazwie *__cplusplus* do kompilacji warunkowej, dzięki czemu w przypadku dołączenia pliku nagłówkowego do kodu kompilowanego kompilatorem języka C++, blok ze specyfikacją linkowania z językiem C znajduje się już w pliku nagłówkowym. Przykład:

```
#ifdef __cplusplus
extern "C"
{
#endif
  /* Treść pliku nagłówkowego biblioteki w języku C */
#ifdef __cplusplus
}
#endif
```

3.3. Przestrzenie nazw

Umieszczanie wszystkich nazw w przestrzeni globalnej może powodować ich konflikty, np. kiedy nadana nazwa została użyta w kilku bibliotekach użytych w jednym projekcie. Język C++ pozwala na rozwiązanie tego problemu poprzez umieszczanie nazw w nazwanych przestrzeniach, w następujący sposób[7]:

```
namespace nazwa_przestrzeni_nazw
```

```
{
  // zawartość przestrzeni nazw
}
```

Dostęp do nazw z przestrzeni możliwy jest na kilka sposobów:

- Za pomocą operatora zasięgu:

```
std::string s;
```

- Poprzez użycie nazwy z przestrzeni:

```
using std::string;
string s;
```

- Poprzez użycie wszystkich nazw z przestrzeni:

```
using namespace std;
string s;
```

3.4. Funkcje składowe

Język C++ pozwala na umieszczanie funkcji wewnątrz struktur, co pozwala na lepszą organizację kodu, dzięki możliwości ich pojęciowego zgrupowania. Funkcje takie umieszczone są w przestrzeni nazw klasy, co zapobiega ich konfliktom.

Domyślnie funkcje składowe przyjmują niejawny argument w postaci wskaźnika na strukturę, w której są zawarte, dostępnego wewnątrz funkcji, poprzez słowo kluczowe *this* oraz umożliwiającego bezpośredni dostęp do wszystkich składowych. Można temu zapobiec poprzez użycie słowa kluczowego static. Aby funkcja składowa mogła operować na stałym obiekcie, należy dodać modyfikator *const* za listą przyjmowanych przez nią argumentów.

Uczynienie funkcji składową klasy, wymaga umieszczenia jedynie jej deklaracji wewnątrz deklaracji klasy. Definicja funkcji może znajdować się na zewnątrz. W takim przypadku, przy jej tworzeniu należy pamiętać, że jej nazwa znajduje się wewnątrz przestrzeni nazw struktury. W przypadku umieszczenia definicji funkcji wewnątrz deklaracji klasy, funkcja staje się domyślnie funkcją *inline*[7].

Pewną wadą funkcji składowych, której nie posiadają normalne funkcje, przyjmujące wskaźnik na strukturę, jest konieczność umieszczania pełnej deklaracji struktury wewnątrz pliku nagłówkowego zawierającego deklaracje takiej funkcji.

3.5. Hermetyzacja

C++ dostarcza mechanizm kontroli dostępu do składowych struktur. Pozwala w ten sposób na uniemożliwienie użytkownikowi bezpośredniego korzystania ze składowych klasy będących detalami implementacyjnymi, przy udostępnieniu publicznego interfejsu. Dzięki temu możliwa jest zmiana implementacji klasy nie wymagająca zmian w korzystającym z niej kodzie[7].

Kontrola ta jest możliwa dzięki udostępnieniu przez język trzech modyfikatorów dostępu:

private:

Składowe prywatne, dostępne dla innych funkcji składowych tej samej klasy oraz klas i funkcji zaprzyjaźnionych.

protected:

Składowe chronione, dostępne dla innych funkcji składowych tej samej klasy, klas dziedziczących po niej oraz klas i funkcji zaprzyjaźnionych.

public:

Składowe publiczne, dostępne wszędzie.

Modyfikator, wraz z dwukropkiem, umieszcza się przed składowymi, których ma dotyczyć. Ze względu na zgodność z językiem C, domyślnie wszystkie składowe struktur są publiczne, jednak C++ wprowadza dodatkowo słowo kluczowe *class*, które od *struct* różni się tylko tym, że składowe klasy są domyślnie prywatne.

Abu uczynić funkcję lub strukturę zaprzyjaźnioną, należy wewnątrz deklaracji klasy umieścić jej deklaracje, poprzedzoną słowem kluczowym *friend*[5].

Ponieważ deklaracje wszystkich funkcji, mających dostęp do składowych prywatnych, znajdują się wewnątrz deklaracji klasy, są łatwe do zlokalizowania, w przypadku, w którym chcielibyśmy zmienić prywatną implementację klasy.

Za przykład klasy posiadającej publiczny interfejs i prywatną implementację może posłużyć klasa przechowująca pozycję punktu w przestrzeni dwuwymiarowej:

```
class point
{
public:
  double x() const { return _x; }
  double y() const { return _y; }
  void x(double x) { _x = x; }
  void y(double y) { _y = y; }
private:
  double _x, _y;
};
```

W obecnej wersji klasa przechowuje pozycję punktu w postaci współrzędnych kartezjańskich w zmiennych prywatnych oraz umożliwia dostęp do nich w takiej postaci, z użyciem interfejsu publicznego. Dzięki takiemu rozwiązaniu możliwa jest zmiana jej implementacji, np. tak aby przechowywała współrzędne w postaci biegunowej, a funkcje implementujące interfejs publiczny dokonywały odpowiednich przeliczeń, bez zmian w samym interfejsie, a co za tym idzie w kodzie korzystającym z powyższej klasy.

3.6. RAII - Inicjowanie przy pozyskaniu zasobu

C++ dostarcza mechanizm konstruktorów i destruktorów wywoływanych automatycznie podczas odpowiednio przydzielania i zwalniania pamięci obiektu. Dzięki jego poprawnemu użyciu otrzymujemy gwarancje, że przydzielony obiekt jest zawsze zainicjowany, a przy usuwaniu zostanie poprawnie zwolniony.

Konstruktor jest specjalną funkcją składową, o nazwie takiej jak klasa, której deklaracja pomija typ wartości zwracanej. Konstruktory mogą być przeciążane na podstawie przyjmowanych parametrów. Kilka z nich posiada specjalne właściwości[5]:

Konstruktor domyślny Konstruktor, który może być wywołany bez podania żadnych argumentów. Wywoływany jest m.in. kiedy lista parametrów konstruktora nie została podana przy tworzeniu obiektu. Jeżeli klasa nie zawiera jawnie zadeklarowanych

innych konstruktorów, konstruktor domyślny dla klasy zostaje wygenerowany automatycznie.

Konstruktor kopiujący Konstruktor kopiujący to konstruktor, którego jedynym, nieposiadającym wartości domyślnej parametrem jest referencja na obiekt tego samego typu lub referencja na stały obiekt tego samego typu. Konstruktor ten jest wywoływany niejawnie przez kompilator, kiedy konieczne jest skopiowanie obiektu, m.in. podczas przekazywania obiektu do funkcji przez wartość. Jeżeli konstruktor kopiujący nie zostanie jawnie zdefiniowany, zostanie wygenerowany automatycznie.

Konstruktor konwertujący Konstruktor posiadający jeden argument, nieposiadający wartości domyślnej może zostać użyty do niejawnej konwersji zmiennej typu takiego jak argument do instancji zawierającej go klasy. Aby temu zapobiec, jego deklarację należy poprzedzić słowem kluczowym *explicit*.

Destruktor jest specjalną funkcją składową, o nazwie składającej się ze znaku tyldy i nazwy klasy. Jego deklaracja, tak jak w przypadku konstruktorów, pomija typ wartości zwracanej[5].

Podczas tworzenia klas według wzorca RAII warto zwrócić uwagę również na automatycznie generowany operator przypisania.

3.6.1. RAII a dynamiczny przydział pamięci

Język C++ dostarcza wsparcia dla dynamicznego zarządzania pamięcią, poprzez dostarczenie operatorów new, new[] oraz delete i delete[]. Użycie ich w surowej wersji wiąże się jednak z kilkoma problemami.

O ile wywołanie operatora delete w celu zwolnienia pamięci przydzielonej obiektowi gwarantuje wywołanie destruktora, to w przypadku użycia gołych wskaźników nie mamy gwarancji, że operator delete w ogóle zostanie wywołany. Rozwiązaniem tego problemu są tzw. inteligentne wskaźniki, czyli obiekty przechowujące wskaźnik na dynamicznie utworzony obiekt, dostarczające interfejs podobny do zwykłych wskaźników oraz destruktor gwarantujący zwolnienie zasobów.

Choć przypisanie wartości zwróconej przez operator new do poprawnego inteligentnego wskaźnika gwarantuje wykonanie operatora delete, należy pamiętać, że w programach

mogą pojawić się błędy na skutek tego, że do takiego przypisania w ogóle nie dojdzie.

```
do_something( smart_pointer( new int(1) ), smart_pointer( new int(2) ) );
```

Na skutek niezdefiniowanej kolejności ewaluacji argumentów funkcji, może zdarzyć się, że pamięć dla jednego z obiektów typu int zostanie poprawnie przydzielona, następnie podczas przydzielania pamięci dla drugiego wystąpi błąd. Inteligentne wskaźniki nie zostaną w takim przypadku utworzone. Rozwiązaniem tego problemu jest użycie fabryk zwracających inteligentny wskaźnik.

Choć większość bibliotek zawierających inteligentne wskaźniki, zawiera również ich wersje do pracy z dynamicznie przydzielanymi tablicami, to poza szczególnymi okolicznościami, lepszym rozwiązaniem jest użycie kontenerów, np. wchodzącego w skład biblioteki standardowej *std::vector*[1].

3.7. Dziedziczenie

Dziedziczenie pozwala na użycie interfejsu oraz implementacji klasy podstawowej, w rozszerzającej ją o dodatkową funkcjonalność klasie pochodnej.

Język C++ pozwala na dziedziczenie po jednej lub wielu klasach oraz specyfikowanie dostępu do klasy podstawowej, tak jak w przypadku zwykłych składowych klasy. Aby utworzyć klasę, będącą pochodną jednej lub wielu klas, w jej deklaracji, po nazwie należy umieścić, poprzedzoną dwukropkiem, rozdzieloną przecinkami listę klas, po których chcemy dziedziczyć. Nazwa każdej z klas podstawowych może być opcjonalnie poprzedzona specyfikatorem dostępu oraz słowem kluczowym *virtual*. Wspomniane słowo kluczowe służy temu, aby w przypadku wielodziedziczenia po klasach posiadających wspólną klasę pochodną, klasy te używały wspólnej instancji wspólnej klasy podstawowej[7].

3.8. Polimorfizm

C++ dostarcza mechanizmu metod wirtualnych, umożliwiających tzw. późne wiązanie, czyli wybór odpowiedniej wersji metody do wykonania w czasie wykonywania, na podstawie typu obiektu. Jest to jeden z mechanizmów pozwalających na oddzielenie interfejsu od implementacji.

Metoda jest wirtualna, jeśli jej deklaracja zostanie poprzedzona słowem kluczowym virtual lub jeśli jest zastępuje implementacje metody wirtualnej z klasy bazowej. Gdy do wskaźnika lub referencji na klasę bazową zostanie przypisana odpowiednio wartość wskaźnika lub referencja na klasę pochodną oraz wywołana zostanie poprzez niego metoda będąca wirtualną w klasie bazowej, wywołana zostanie jej wersja z klasy pochodnej, nawet jeśli właściwy typ nie był znany podczas kompilacji.

Kolejną cechą języka są funkcje czysto wirtualne. Są to funkcje wirtualne, nie posiadające implementacji w danej klasie. Aby uczynić funkcję czysto wirtualną, w jej definicji blok kodu należy zastąpić przypisaniem zera.

Klasy zawierające funkcje czysto wirtualne nazywamy klasami abstrakcyjnymi. Nie można stworzyć ich instancji, a jedynie wskaźnik lub referencję do nich[7].

Podczas implementacji polimorficznych typów należy zwrócić szczególną uwagę na destruktory. Obiekty takie bardzo często tworzone są w dynamicznie przydzielanej pamięci, a wskaźniki do nich rzutowane są na wskaźniki do klasy bazowej, tak że kod ich używający nie posiada informacji o typie obiektu. Wywołanie operatora delete na takim wskaźniku gwarantuje prawidłowe zwolnienie pamięci, jednak jeżeli destruktor klasy podstawowej nie jest wirtualny, destruktor klasy pochodnej nie zostanie wywołany[5].

3.9. Programowanie uogólnione

Programowanie uogólnione polega na tworzeniu elementów oprogramowania nadających się do ponownego użycia w różnych sytuacjach. C++ dostarcza mechanizmu szablonów, umożliwiającego generalizację klas i funkcji bez poświęcania wydajności. Szablony mogą być parametryzowane typami oraz liczbami całkowitymi znanymi w czasie kompilacji[3].

Aby utworzyć szablon klasy lub funkcji, jej deklarację należy poprzedzić słowem kluczowym *templat*e oraz oddzieloną przecinkami listą parametrów szablonu. Definicje szablonów klas i funkcji muszą znajdować się w jednostce translacji, w której są używane. W tym celu umieszczane są zazwyczaj w plikach nagłówkowych[7]. Poniższy przykład zawiera definicje szablonu klasy opakowującej tablicę:

```
template<typename T, size_t size>
class array
```

```
{
public:
  T& operator[] (size_t n) { return array[n]; }
  const T& operator[] (size_t n) const { return array[n]; }
private:
  T array[size];
};
```

W programowaniu uogólnionym ważną rolę pełnią koncepty, podobne do interfejsów w programowaniu obiektowym. Koncept jest zbiorem poprawnych wyrażeń, powiązanych typów, niezmienników oraz gwarancji co do złożoności obliczeniowej i pamięciowej. Typ spełniający wymagania konceptu nazywamy jego modelem[3].

3.10. Wyjątki

Język C++ zawiera, służący obsłudze błędów, mechanizm wyjątków. Pozwala on na oddzielenie kodu obsługi błędów, od kodu realizowanego w pożądanej sytuacji. Kolejną cechą tego mechanizmu jest to, że zgłoszony błąd nie może zostać zignorowany. Kiedy nie zostanie obsłużony w funkcji w której wystąpił, zostanie przerzucony niżej, gdzie może zostać obsłużony z wykorzystaniem szerszych informacji o kontekście. Jeśli nie zostanie nigdzie obsłużony, spowoduje wywołanie odpowiedniej funkcji, domyślnie kończącej wykonywanie programu *std::terminate*. Wyjątek może być obiektem dowolnego typu.

Aby rzucić wyjątek, należy użyć słowa kluczowego *throw*, a za nim obiektu, który ma zostać rzucony. Zazwyczaj jest to obiekt tymczasowy, typu stworzonego specjalnie do obsługi tego rodzaju błędów.

Aby przechwycić wyjątek, wyrzucający go kod należy umieścić wewnątrz bloku, poprzedzonego słowem kluczowym *try*, za którym umieszczone zostają procedury obsługi wyjątków. Procedura obsługi wyjątków składa się z bloku poprzedzonego słowem kluczowym *catch* oraz nawiasami zawierającymi typ wyjątku obsługiwanego przez tą procedurę oraz opcjonalną nazwę zmiennej, umożliwiającą dostęp do obiektu wyjątku, wewnątrz procedury jego obsługi lub trzy kropki, jeżeli procedura ma przechwytywać wyjątki dowolnego typu.

C++ umożliwia specyfikowanie typów wyjątków jakie mogą zostać wyrzucone z funkcji. Aby tego dokonać na końcu deklaracji funkcji należy umieścić słowo kluczowe *thro*w oraz oddzieloną przecinkami listę typów wyjątków, jakie mogą zostać wyrzucone z funkcji. Brak takiej specyfikacji oznacza, że funkcja może wyrzucić wyjątek dowolnego typu[8].

Wyróżnia się trzy poziomy gwarancji bezpieczeństwa kodu, jeśli chodzi o użycie wyjątków[6]:

Podstawowa gwarancja

Niezmienniki komponentu nie są naruszone oraz nie nastąpił wyciek zasobów

Silna gwarancja

W przypadku rzucenia wyjątku program znajduje się w stanie takim jak przed rozpoczęciem operacji

Gwarancja nie rzucenia wyjątku

Operacja nigdy nie powoduje rzucenia wyjątku

4. Budowanie zależności

4.1. igraph

Biblioteka igraph, w wydanej 17 Czerwca 2012 roku wersji 0.6, została zbudowana na użytek niniejszej pracy, z użyciem poniższych komend:

```
tar xvf igraph-0.6.tar.gz
cd igraph-0.6/
CFLAGS="-O3 -DNDEBUG -s -fexceptions" ./configure
make
```

W bieżącym katalogu znajdowały się katalog *include*, zawierający pliki nagłówkowe biblioteki oraz *src/.libs*, zawierający zbudowaną bibliotekę.

4.2. Boost

Biblioteki Boost, w wersji 1.51.0 wydanej 20 sierpnia 2012 roku, użyte zostały do implementacji tworzonej w ramach niniejszej pracy biblioteki, jej testów jednostkowych oraz testów porównujących wydajność z interfejsem dla języka C oraz biblioteką igraphpp. Większość z nich składa się wyłącznie z plików nagłówkowych, jednak do działania części konieczne okazało się ich zbudowanie. W tym celu użyte zostały niniejsze komendy:

```
tar xvf boost_1_51_0.tar.bz
cd boost_1_51_0
./bootstrap.sh --with-libraries=timer,system,chrono,test
./b2 variant=release
```

Zgodnie z wyświetlonym komunikatem, potrzebne pliki nagłówkowe znajdowały się w katalogu *include*, natomiast zbudowane biblioteki w */home/witek/inz/boost_1_51_0/stage/lib*.

4.3. Loki

Biblioteka Loki, w wersji 0.1.7 z dnia 29 stycznia 2009 roku, została użyta do implementacji tworzonej biblioteki. Biblioteka została zbudowana przy pomocy niniejszych komend:

```
tar xvf loki-0.1.7.tar.bz2
cd loki-0.1.7
make
```

Potrzebne pliki nagłówkowe znajdowały się w podkatalogu *include*, natomiast zbudowana biblioteka z katalogu *lib*

4.4. igraphpp

Wydajność stworzonego interfejsu została porównana z istniejącym interfejsem języka C++ dla biblioteki igraph o nazwie *igraphpp*, w pochodzącej z systemu kontroli wersji git rewizji *f03616bdd468e465b9789e9b173fb186beecdfff*, z dnia 28 lipca 2012 roku. W celu jej zbudowania posłużono się następującymi komendami:

```
cd igraphpp
mkdir build
cd build
cmake -DCMAKE\_BUILD\_TYPE="Release" ..
make
```

Pliki nagłówkowe, tak jak w poprzednich przypadkach, znajdowały się w podkatalogu *include* głównego katalogu z biblioteką, natomiast zbudowana biblioteka, tylko w wersji do linkowania statycznego, w jego podkatalogu *build/src*.

Warto wspomnieć, że zbudowanie biblioteki igraphpp w trybie innym niż Release nie udało się ze względu na niezdefiniowany symbol wewnątrz makra *assert*.

5. Interfejs stworzonej biblioteki

5.1. Biblioteka tylko nagłówkowa

Biblioteka igraphxx została zaimplementowana w całości wewnątrz plików nagłówkowych, dzięki czemu jej użycie nie wymaga budowania oraz linkowania, a jedynie dołączenia plików nagłówkowych.

Podczas tworzenia tego typu bibliotek, pewną trudność stanowić może pozostanie w zgodzie z regułą jednej definicji, ponieważ pliki nagłówkowe mogą zostać dołączone w kilku jednostkach translacji. W niniejszej pracy użyto dwóch sposobów rozwiązania tego problemu:

- Skorzystanie z wyjątków od tej reguły, do których należą definicje funkcji *inline*, funkcji szablonowych oraz funkcji składowych klas szablonowych.
- Umieszczenie symboli biblioteki w nienazwanej przestrzeni nazw. Należy pamiętać, że w poszczególnych jednostkach translacji są to różne symbole.

5.2. Struktury danych

Jedyną strukturą danych biblioteki igraph użytą w stworzonym interfejsie jest wektor liczb rzeczywistych. Opakowująca go klasa *vector* jest modelem pochodzących z biblioteki *STL* konceptów *RandomAccessContainer* oraz *BackInsertionSequence*. Klasa ta używana jest do jednoczesnego dodawania wielu krawędzi do grafu, może także być użyta do przechowywania atrybutów wierzchołków i krawędzi. Dzięki możliwości pobrania wskaźnika do typu *igraph_vector_t* może być użyta również z algorytmami interfejsu dla języka C, np. do podania atrybutu jako wagi wierzchołków lub krawędzi.

5.3. Selektory oraz iteratory

Selektory biblioteki igraph zostały opakowane w klasy *vertices_selector* oraz *edge_selector*. Zaimplementowano jedynie konstruktory, po użyciu których, selektor nie staje się właścicielem kolekcji etykiet wierzchołków lub krawędzi. Dzięki temu istnieje możliwość przekazywania ich jako zmiennych tymczasowych. Własność ta nie wynika jednak z dokumentacji biblioteki igraph, więc istnieje ryzyko, że ulegnie zmianie w kolejnych jej wersjach.

W stworzonej bibliotece znalazły się klasy *vertices_iterator* oraz *edges_iterator* adaptujące iteratory biblioteki igraph, na iteratory biblioteki STL. Ze względu na ubogi interfejs adaptowanych iteratorów, są one modelem najprostszego konceptu iteratora - *Input Iterator*. Wspomniane klasy przyjmują jako argument konstruktora wskaźnik do iteratorów interfejsu dla języka C, nie przejmując własności. Z tego powodu najlepiej używać ich wraz z odpowiadającymi za nią typami *vertices_range* i *edges_range*, będącymi modelami konceptu *SinglePassRange* z biblioteki Boost.

5.4. Graf

Biblioteka igraphxx zawiera klasę *graph*, odpowiadającą za własność obiektów struktury reprezentujące graf w bibliotece igraph oraz dostarczającą interfejsu do jego modyfikacji, zliczania wierzchołków i krawędzi oraz pobierania ogona i głowy krawędzi, na podstawie jej etykiety i odwrotnie.

5.5. Obsługa błędów

Biblioteka igraphxx zawiera system obsługi błędów dla biblioteki igraph, wykorzystujący mechanizm wyjątków języka C++. System ten wymaga, aby funkcje biblioteki *igraph* potrafiły rzucić dalej rzucony do nich wyjątek. Jeżeli do budowy biblioteki igraph używa się kompilatora języka C wchodzącego w skład GNU Compiler Collection, w celu spełnienia tego wymagania należy użyć opcji *-fexceptions*.

Typy rzucanych wyjątków są konkretyzacjami szablonu parametryzowanego kodem błędu biblioteki igraph. Dziedziczą one po wspólnej klasie *exception*, dziedziczącej po klasie o tej samej nazwie z biblioteki STL.

5.6. Atrybuty

Stworzona biblioteka zawiera niepełną implementację systemu pozwalającego na przechowywanie obiektów języka C++ dowolnego typu jako atrybutów wierzchołków i krawędzi grafów biblioteki igraph, wewnątrz dowolnego kontenera będącego modelem konceptów *RandomAccessContainer* i *Sequence*.

Mimo iż obecna wersja nie zawiera wszystkich wymaganych przez bibliotekę igraph funkcji, istniejący zestaw pozwala na jej użycie z prawie wszystkimi funkcjami biblioteki igraph.

6. Testy jednostkowe

6.1. O testowaniu jednostkowym

Testowanie jednostkowe polega na próbie wykazania niezgodności modułów programu, takich jak funkcje, czy biblioteki dynamiczne, ze specyfikacją. Dokonuje się tego poprzez sprawdzenie ich działania wraz z dobranymi w tym celu przypadkami testowymi[9].

Testowanie oprogramowania rozpoczyna się zazwyczaj od testowania jednostkowego z następujących powodów[9]:

- Małe fragmenty kodu są mniej skomplikowane w testowaniu, co przyczynia się do zwiększenia efektywności testowania.
- Wykrycie błędu w konkretnym, małym module, ułatwia znalezienie jego przyczyny.
- Niezależne moduły można testować równolegle.

6.2. Testowanie przyrostowe wstępujące

Podczas implementacji testów jednostkowym, dla stworzonej w ramach niniejszej pracy biblioteki, zastosowane zostało testowanie przyrostowe wstępujące. Polega ono na rozpoczęciu testów od modułów niezależących od innych modułów, w następnej kolejności modułów zależnych tylko od tych które zostały już przetestowane i tak w kółko, aż przetestowane zostaną wszystkie moduły. Rozwiązanie takie możliwe jest tylko jeśli pomiędzy modułami nie występują cykliczne zależności.

Wybrany sposób organizacji testów pozwala na uniknięcie tworzenia namiastek zależności testowanych modułów. Testowanie przyrostowe, w przeciwieństwie do testowania rozproszonego, pozwala ponadto na wykrycie błędów polegających na niezgodności testowanych modułów. Wiąże się z nim jednak ograniczona możliwość zrównoleglenia testów[9].

6.3. Dobór przypadków testowych

6.3.1. Testy białej skrzynki

Testowanie metodą białej skrzynki opiera się na znajomości testowanego kodu[9]. Większość funkcji zaimplementowanych w stworzonej bibliotece posiada tylko jedną możliwą ścieżkę przepływu, ponieważ ogranicza się do wywołania swojego odpowiednika z interfejsu dla języka C oraz ewentualnego przekazania mu parametrów oraz zwrócenia odpowiedzi. Jedynymi rozgałęzieniami w tego typu funkcjach mogą być zdarzenia zainicjowane przez używany system obsługi błędów, jednak zostały one zignorowane z kilku powodów:

- Reakcja na błąd może być różna w zależności od użytego systemu obsługi błędów.
- Możliwość wystąpienia błędu nie powinna być obsługiwana przez tego typu funkcje.
- System obsługi błędów został przetestowany niezależnie.
- Celowe wywołanie części błędów byłoby kłopotliwe.
- Dokumentacja biblioteki igraph nie wymienia możliwych typów błędów w przypadku części funkcji.

Z wymienionych powodów testowanie wspomnianych metodą białej skrzynki ograniczało się do użycia jednego, dowolnego przypadku testowego.

W przypadku bardziej złożonych funkcji, np. wchodzących w skład systemu obsługi atrybutów starano się spełnić kryterium pokrycia wielowarunkowego. Polega ono na przetestowaniu wszelkich kombinacji składowych wyrażenia decyzyjnego[9].

6.3.2. Testy czarnej skrzynki

Testowanie metodą czarnej skrzynki polega na wybraniu przypadków testowych w oparciu o specyfikacje testowanej jednostki, nie wnikając w sposób jej implementacji[9]. Podczas implementacji testów jednostkowych w ramach niniejszej pracy starano się wykorzystać dwie techniki należące do tej metody testowania:

Podział na klasy równoważności Technika polegająca na podzieleniu danych wejściowych testowanej jednostki na klasy, w których przetestowanie jednej wartości powoduje silne obniżenie prawdopodobieństwa wykrycia błędu poprzez przetestowanie innej wartości z tej samej klasy[9].

Analiza wartości granicznych Projektowanie przypadków testowych w oparciu o dane leżące na granicach klas równoważności[9].

6.4. Platforma Boost Test

Biblioteka Boost Test dostarcza zestawu narzędzi wspomagającego implementację testów oprogramowania oraz ich organizację w moduły, pakiety i przypadki testowe. Domyślnie każdy moduł jest oddzielną jednostką translacji zawierającą funkcję main, więc budowany jest do oddzielnego pliku wykonywalnego. Każdy z modułów może zostać podzielony na dowolną liczbę nazwanych pakietów, które z kolei dzieli się na nazwane przypadki testowe, zawierające dowolną liczbę warunków.

Do implementacji warunków warto używać odpowiednio udostępnionych przez bibliotekę makr, takich jak *BOOST_CHECK*, czy *BOOST_CHECK_EQUAL*, które usprawniają wyświetlani komunikatów informujących o ewentualnych niepowodzeniach.

Biblioteka dostarcza również mechanizmu wspomagającego inicjację i sprzątanie danych używanych w testach globalnie, w pakiecie lub przypadku testowym[1].

Podczas implementacji testów w ramach niniejszej pracy przyjęto następujące założenia:

- Moduł testów odpowiada klasie stworzonej biblioteki.
- Każdy pakiet odpowiada funkcji należącej do klasy lub związanej z nią w inny sposób.
- Przypadek testowy odpowiada jednemu zestawowi danych testowych.

6.5. Wnioski

W ramach niniejszej pracy nie udało się osiągnąć zakładanego pokrycia testami. Spowodowane jest to najprawdopodobniej przyjęciem zbyt ostrych kryteriów pokrycia kodu,

przy ilości włożonego czasu. Kolejnymi błędami okazało się odkładanie stworzenia testów w czasie oraz implementowanie ich w oparciu o niestarannie dobrane przypadki testowe oraz planowanie ich ulepszenia w przyszłości.

Gdyby przyjąć lżejsze kryteria pokrycia kodu łatwiej byłoby zaimplementować testy odpowiednio wcześniej, być może nawet przed implementacją samej biblioteki oraz z należytą starannością. To z kolei prawdopodobnie pozwoliło by zaoszczędzić sporo czasu na lokalizowaniu błędów ujawniających się już podczas tworzenia biblioteki, uniknięciu części problemów związanych z projektem interfejsu oraz staranniejsze zaprojektowanie przypadków testowych, a co za tym idzie również lepsze przetestowanie stworzonej biblioteki.

7. Testy wydajności

7.1. Środowisko pracy

Testy wydajności zostały uruchomione na komputerze Toshiba Satellite L300-11F. Komputer ten wyposażony jest w procesor Intel Celeron 550, taktowany zegarem 2 GHz oraz 1 GB pamięci RAM typu DDR2. Komputer pracował pod kontrolą systemu operacyjnego Debian GNU/Linux w wersji 6.0 Squeeze. Do zbudowania testów oraz zależności wykorzystane zostały, wchodzące w skład GNU Compiler Collection w wersji Debian 4.4.5-8, kompilatory języków C i C++, pochodzące z oficjalnego repozytorium pakietów użytego systemu.

7.2. Wyniki

7.2.1. Inicjacja i zwalnianie grafu

Czas wykonania 100 000 operacji w milisekundach

	interfejs dla języka c	**igraphxx**	**igraphpp**
Najmniejszy	53,9	54,1	60,7
Średni	54,7	54,7	62,3
Największy	81,2	58,4	78,6

Wyniki osiągnięte przez interfejs dla języka C i zaimplementowaną bibliotekę różnią się nieznacznie. Narzut podczas użycia biblioteki igraphpp wynika prawdopodobnie głównie z użycia pamięci dynamicznej wewnątrz klasy *Graph* oraz destruktora niewstawianego w miejscu wywołania.

7.2.2. Dodanie wierzchołka do grafu

Czas wykonania 100 000 operacji w milisekundach

	interfejs dla języka c	**igraphxx**	**igraphpp**
Najmniejszy	17,1	17,1	17,5
Średni	17,7	17,7	18,2
Największy	20,8	20,8	23,4

Biblioteka igraphxx nie wprowadziła narzutu podczas operacji dodawania wierzchołków. Narzut biblioteki igraphpp wynika prawdopodobnie z niewstawienia funkcji w miejsca wywołania oraz dostępu do struktury grafu za pośrednictwem wskaźnika.

7.2.3. Dodanie krawędzi do grafu

Czas wykonania 100 operacji w milisekundach

	interfejs dla języka c	**igraphxx**	**igraphpp**
Najmniejszy	1	1	1
Średni	1,3	1	1,1
Największy	5,1	1,1	1,8

Operacja okazała się na tyle wymagająca obliczeniowo, że wpływ wstawiania funkcji w miejscu wywołania oraz dostępu za pośrednictwem wskaźnika okazał się pomijalny.

7.2.4. Kopiowanie i niszczenie pustego grafu

Czas wykonania 100 000 operacji w milisekundach

	interfejs dla języka c	**igraphxx**	**igraphpp**
Najmniejszy	56,1	56	65,3
Średni	57,1	57,3	67,9
Największy	68,6	73,9	77,8

Zaimplementowana biblioteka nie wprowadziła dodatkowego narzutu. W przypadku biblioteki igraphpp narzut wynika prawdopodobnie głównie z użycia dynamicznej alokacji pamięci.

7.2.5. Usunięcie jednego ze 100 wierzchołków z pełnego grafu

Czas wykonania 10 operacji w milisekundach

	interfejs dla języka c	**igraphxx**
Najmniejszy	16,9	16,9
Średni	19,2	24,4
Największy	78,3	87,8

Brak zauważalnego narzutu wprowadzonego przez zaimplementowaną bibliotekę. Biblioteka igraphpp nie została przetestowana, ponieważ w jej interfejsie nie znaleziono odpowiedniej funkcji.

7.2.6. Dodawanie napisu jako atrybutu wierzchołka

Czas wykonania operacji w milisekundach na grafie o 100 000 wierzchołkach

	interfejs dla języka c	**igraphxx**	**igraphpp**
Najmniejszy	12,6	4,8	21,7
Średni	13,2	7,1	28,9
Największy	67,8	25,6	87,7

Najprawdopodobniej główną przyczyną tak znacznej przewagi zaimplementowanej biblioteki jest dostęp do kontenera przechowującego wartości atrybutów poszczególnych wierzchołków po jednorazowym podaniu nazwy atrybutu. Pozostałe biblioteki wymagają podania nazwy podczas pojedynczego przypisania wartości do każdego z wierzchołków.

7.2.7. Kopiowanie grafu posiadającego tekstowy atrybut wierzchołka

Czas wykonania operacji w milisekundach na grafie o 100 000 wierzchołkach

	interfejs dla języka c	**igraphxx**	**igraphpp**
Najmniejszy	10,5	3,4	13,6
Średni	11	3,4	14,19
Największy	12	5,4	18,2

Zaimplementowana biblioteka uzyskała przewagę nad biblioteką igraph dzięki użyciu własnego systemu przechowywania atrybutów, opartego o kontenery biblioteki *STL*. Przyczyną słabej wydajności biblioteki igraphpp jest prawdopodobnie przechowywanie wartości atrybutów z użyciem typu *any*.

7.2.8. Obsługa błędów

Sprawdzanie, czy wystąpił błąd - nie wystąpił, 1 000 000 razy

	interfejs dla języka c	**igraphxx**
Najmniejszy	70,8	70,3
Średni	72,1	71,4
Największy	75,9	75,5

Sprawdzanie, czy wystąpił błąd - wystąpił, 10 000 razy

	interfejs dla języka c	**igraphxx**
Najmniejszy	0,2	103,8
Średni	0,2	104,8
Największy	0,3	121,6

Jak widać, podczas pomyślnego przebiegu programu wydajność obsługi błędów za pomocą zwracania kodu błędu i za pomocą wyjątków nie odbiega od siebie. Z kolei podczas wystąpienia błędów, wykrycie tego faktu za pomocą kodu powrotu jest kilkaset razy szybsze.

8. Generowanie dokumentacji

Dokumentacja biblioteki igraphxx została stworzona z wykorzystaniem programu Doxygen. Doxygen pozwala na generowanie dokumentacji na podstawie kodu źródłowego oraz umieszczonych w nim specjalnych komentarzy. Komentarze mogą służyć zarówno opisaniu elementów kodu źródłowego, uzupełniając generowaną na jago podstawie dokumentacje o dodatkowe informacje, jak i tworzyć dokumentacje o niezależnej od kodu strukturze.

Doxygen obsługuje różne języki programowania, w skład których wchodzi C++ oraz pozwala na generowanie dokumentacji w różnych formatach, w skład których wchodzą m.in. HTML, LaTeX, czy strony podręcznika systemu Unix[2].

W ramach niniejszej pracy stworzone zostały komentarze, w których opisano wchodzące w skład stworzonego interfejsu przestrzenie nazw, klasy, wybrane funkcje, pliki nagłówkowe oraz przykłady. Dodano również komentarze tworzące kilka niezależnych od kodu stron, w treści których, umieszczono ogólny opis biblioteki. Następnie wygenerowano dokumentację w formacie HTML.

9. Przykłady

9.1. Tworzenie i modyfikowanie grafu

```
#include <igraphxx/basic.hpp>
#include <igraphxx/vector_assign.hpp>
#include <iostream>

using namespace igraphxx::assign;

void print_graph(igraphxx::graph& graph)
{
  igraphxx::vertices_selector vs( igraphxx::vertices_selector::
     all() );
  igraphxx::edges_selector es( igraphxx::edges_selector::all() )
     ;
  std::cout << "Graph contain " << graph.count_vertices( vs ) <<
      " vertices and "
            << graph.count_edges( es ) << " edges:";

  igraphxx::edges_range all_edges(graph, es);
  for(
      igraphxx::edges_iterator it( all_edges.begin() );
      it != all_edges.end();
      ++it
      )
    {
      std::cout << " (" << graph.edge(*it).first << ", " <<
```

```
            graph.edge(*it).second << ")";
        }
    std::cout << std::endl;
}

int main()
{
    std::cout << "Creating graph with 5 vertices, and edges from 0
        to any other..." << std:: endl << "\n";
    igraphxx::graph some_graph(false, 5);
    {
        igraphxx::vector edges;
        edges += 0,1, 0,2, 0,3, 0,4;
        some_graph.add_edges(edges);
    }
    print_graph(some_graph);

    std::cout << "Removing vertices adjacent to 0..." << std::endl
        << "\n";
    igraphxx::vertices_selector neighbours_of_0( igraphxx::
        vertices_selector::adjacent(0) );
    some_graph.remove_vertices(neighbours_of_0);
    print_graph(some_graph);

    std::cout << "Adding 2 vertices and connect as ring..." << std
        ::endl << "\n";
    some_graph.add_vertices(2);
    some_graph.add_edge(0, 1);
    some_graph.add_edge(1, 2);
    some_graph.add_edge(2, 0);
    print_graph(some_graph);
```

```
  std::cout << "Removing edge from 2 to 1..." << std::endl << "
     ";
  igraphxx::edge_key key_2_to_1( some_graph.edge(2, 1) );
  igraphxx::edges_selector es_2_to_1( key_2_to_1 );
  some_graph.remove_edges(es_2_to_1);
  print_graph(some_graph);
}
```

9.2. Obsługa błędów przy użyciu wyjątków

```
#include <igraphxx/basic.hpp>
#include <igraphxx/error_handling.hpp>
#include <iostream>

using namespace igraphxx;
using namespace std;

int main()
{
  ::igraph_set_error_handler(igraphxx::error_handler);
  graph graph(true, 1);
  for(;;)
    {
      try
        {
          vertices_selector all_vertices( vertices_selector::all
             () );
          integer vertices_count( graph.count_vertices(
             all_vertices) );
          graph.add_vertices(1);
          while(vertices_count > 0)
```

```
            {
              graph.add_edge(vertices_count, vertices_count -1);
              --vertices_count;
            }
        }
      catch(igraphxx::errors::enomem&)
        {
          std::cout << "Can't add more vertices and edges: Out
              of memory." << std::endl;;
          break;
        }
      catch(std::exception& e)
        {
          std::cout << "Can't add more vertices and edges: " <<
              e.what() << std::endl;
          break;
        }
    }
  vertices_selector all_vertices( vertices_selector::all() );
  edges_selector all_edges( edges_selector::all() );
  std::cout << "This program can work yet with " << graph.
      count_vertices(all_vertices) << " and " << graph.
      count_edges(all_edges) << " edges." << endl;
}
#include <igraphxx/basic.hpp>
#include <igraphxx/error_handling.hpp>
#include <iostream>

using namespace igraphxx;
using namespace std;
```

```
int main()
{
  ::igraph_set_error_handler(igraphxx::error_handler);
  graph graph(true, 1);
  for(;;)
    {
      try
        {
          vertices_selector all_vertices( vertices_selector::all
              () );
          integer vertices_count( graph.count_vertices(
              all_vertices) );
          graph.add_vertices(1);
          while(vertices_count > 0)
            {
              graph.add_edge(vertices_count, vertices_count -1);
              --vertices_count;
            }
        }
      catch(igraphxx::errors::enomem&)
        {
          std::cout << "Can't add more vertices and edges: Out
              of memory." << std::endl;;
          break;
        }
      catch(std::exception& e)
        {
          std::cout << "Can't add more vertices and edges: " <<
              e.what() << std::endl;
          break;
        }
```

```
    }
  vertices_selector all_vertices( vertices_selector::all() );
  edges_selector all_edges( edges_selector::all() );
  std::cout << "This program can work yet with " << graph.
      count_vertices(all_vertices) << " and " << graph.
      count_edges(all_edges) << " edges." << endl;
}
```

9.3. Przypisanie atrybutów do wierzchołków i krawędzi

```
#include <igraphxx/basic.hpp>
#include <igraphxx/attributes.hpp>
#include <igraphxx/vector_assign.hpp>
#include <iostream>
#include <vector>
#include <string>
#include <iterator>
#include <algorithm>

using namespace igraphxx::assign;

int main()
{
  ::igraph_i_set_attribute_table(&igraphxx::attribute_table);

  igraphxx::graph graph1(false, 5);
  {
    igraphxx::vector edges;
    edges += 0,1, 1,2 , 2,3, 2,4;
    graph1.add_edges(edges);
```

```
}

igraphxx::vector& vertices_nums( igraphxx::attribute_handler::
    vertices_attribute<igraphxx::vector>(graph1, "num") );
vertices_nums[0] = 0;
vertices_nums[1] = 1;
vertices_nums[2] = 2;
vertices_nums[3] = 3;
vertices_nums[4] = 4;
std::vector< std::string >& edges_endpoints( igraphxx::
    attribute_handler::edges_attribute< std::vector<std::string
   > >(graph1, "vertices") );
edges_endpoints[0] = "(0,1)";
edges_endpoints[1] = "(1,2)";
edges_endpoints[2] = "(2,3)";
edges_endpoints[3] = "(2,4)";

igraphxx::graph graph2(graph1);
graph2.add_vertices(1);
graph2.add_edge(2, 5);
igraphxx::vertices_selector vs0(0);
graph2.remove_vertices(vs0);

igraphxx::vector& graph1_vertices( igraphxx::attribute_handler
    ::vertices_attribute<igraphxx::vector>(graph1, "num") );
std::cout << "graph1 vertices: " << std::endl;
std::copy( graph1_vertices.begin(), graph1_vertices.end(), std
    ::ostream_iterator<long>(std::cout, "\r\n") );
std::vector< std::string >& graph1_edges( igraphxx::
    attribute_handler::edges_attribute< std::vector<std::string
   > >(graph1, "vertices") );
```

```
  std::cout << "graph1 edges: " << std::endl;
  std::copy( graph1_edges.begin(), graph1_edges.end(), std::
     ostream_iterator<std::string>(std::cout, "\r\n") );
  igraphxx::vector& graph2_vertices( igraphxx::attribute_handler
     ::vertices_attribute<igraphxx::vector>(graph2, "num") );
  std::cout << "graph2 vertices: " << std::endl;
  std::copy( graph2_vertices.begin(), graph2_vertices.end(), std
     ::ostream_iterator<long>(std::cout, "\r\n") );
  std::vector< std::string >& graph2_edges( igraphxx::
     attribute_handler::edges_attribute< std::vector<std::string
    > >(graph2, "vertices") );
  std::cout << "graph2 edges: " << std::endl;
  std::copy( graph2_edges.begin(), graph2_edges.end(), std::
     ostream_iterator<std::string>(std::cout, "\r\n") );
}
```

9.4. Użycie algorytmów z interfejsu dla języka C

```
#include <igraphxx/basic.hpp>
#include <boost/bind.hpp>
#include <iostream>

int main()
{
  igraphxx::graph graph(boost::bind(igraph_ring, _1, 100, false,
      false, true));
  igraphxx::vector vec;
  igraph_get_shortest_path(graph, vec.get_ptr(), NULL, 20, 75,
     IGRAPH_ALL);
  std::cout << vec.size() << std::endl;
}
```

10. Podsumowanie

Celem niniejszej pracy było stworzenie interfejsu biblioteki igraph dla języka C++. Cel ten został osiągnięty, poprzez stworzenie biblioteki igraphxx, składającej się wyłącznie z plików nagłówkowych.

Stworzona biblioteka złożona jest z trzech niezależnych części. Pierwsza z nich zawiera klasy opakowujące struktury biblioteki igraph, umożliwiając tworzenie i modyfikowanie grafów, oraz dostęp do ich podstawowych własności. Kolejna umożliwia sygnalizowanie wewnętrznych błędów biblioteki igraph, za pomocą wyjątków języka C++. Dzięki ostatniej z nich, możliwe jest przechowywanie dowolnych obiektów języka C++, jako atrybutów wierzchołków i krawędzi, wewnątrz kontenera, o typie wskazanych przez użytkownika biblioteki.

Dla stworzonej biblioteki zaimplementowano testy jednostkowe, używając biblioteki Boost Test. Niestety ze względu na niedocenienie zarówno roli samego procesu testowania, jak i ilości pracy, jaką należało by włożyć w zaimplementowanie dobrego zestawu testów, nie osiągnięto zadowalającego pokrycia kodu testami. Mimo tego, uruchomienie przykładów wykorzystania biblioteki odbyło się bez problemów, wynikających z błędów w bibliotece.

Wydajność stworzonej biblioteki została porównana z wydajnością interfejsu biblioteki igraph dla języka C, oraz innego interfejsu dla języka C++, zaimplementowanego w postaci biblioteki igraphpp. Testy wykazały, że opakowanie struktur biblioteki igraph w klasy, oraz operujących na nich funkcji w ich metody, nie spowodowało narzutu wydajnościowego. Podczas wykorzystywania części odpowiedzialnej za przechowywanie atrybutów, posiadającej implementację niezależną od swojego odpowiednika z biblioteki igraph, udało się uzyskać znaczący przyrost wydajności, w stosunku do pierwowzoru. Wynika to najprawdopodobniej z użycia generycznych klas, wchodzących w skład dobrze zaprojektowanych bibliotek. Testy wykazały również, że użycie mechanizmu wyjątków, do sygnalizowania błędów, nie spowodowało narzutu wydajnościowego, w stosunku do sprawdzania

kodu błędu zwracanego przez funkcję w przypadku braku błędów.

W ramach niniejszej pracy stworzona została również dokumentacja biblioteki igraphxx. W tym celu wykorzystany został program Doxygen, generujący dokumentację na podstawie kodu źródłowego, oraz umieszczonych w nim komentarzy o specjalnej składni. Bardzo dobrym posunięciem okazało się tworzenie wspomnianych komentarzy już podczas tworzenia kodu źródłowego. Ułatwiło to orientacje w kodzie po przerwach w pracy nad nim, a także podczas pisania pracy.

Literatura

[1] Boost 1.51.0 library documentation. http://www.boost.org/doc/libs/1_51_0/.

[2] Doxygen manual. http://www.stack.nl/~dimitri/doxygen/manual.html.

[3] Generic programming techniques. http://www.boost.org/community/generic_programming.htm

[4] Igraph reference manual. http://igraph.sourceforge.net/doc/python/igraph.pdf.

[5] Iso/iec 14882:1998(e), programming languages ? c++.

[6] David Abrahams. Exception-safety in generic components. http://www.boost.org/community/exception_safety.html.

[7] Bruce Eckel. *Thinking in C++. Edycja polska.* Helion, 2002.

[8] Bruce Eckel and Chuck Allison. *Thinking in C++. Edycja polska. Tom 2.* Helion, 2004.

[9] Glenford J. Myers, Corey Sandler, Tom Badgett, and Todd M. Thomas. *Sztuka testowania oprogramowania.* Helion, 2005.

[10] Bjarne Stroustrup. A history of c++: 1979-1991. http://www.stroustrup.com/hopl2.pdf.