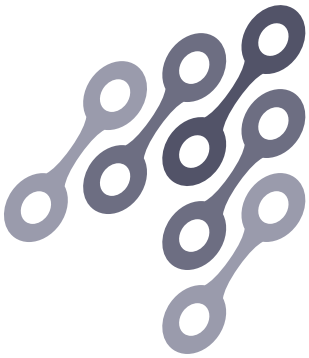
POLITECHNIKA CZĘSTOCHOWSKA
WYDZIAŁ INŻYNIERII MECHANICZNEJ
I INFORMATYKI



PRACA MAGISTERSKA

Implementacja konfiguracji biblioteki grafowej SCGL

Implementation of the SCGL graph library configuration

Autor: Marcin Ptak
Nr indeksu: 115501
Kierunek: Informatyka
Promotor: dr inż. Ireneusz Szcześniak

Praca przyjęta dnia: .....

Podpis promotora: .....

Częstochowa, 2013

Abstract

This master's thesis describes the implementation of the configuration of the SCGL graph library. The main objective of this work was to implement new features and optimise performance of the *Simple C Graph Library*—a minimalistic, open-source graph library written in C language. The methodology adopted for achieving this goal involved both the software engineering methods as well as experimental approach.

The author implemented a configuration feature in the SCGL library using CMake, an open-source build tool. Configuration feature allows to build a library tailored to needs and therefore can make it more efficient. For instance, configuration of the SCGL library introduces 'capacity'—an additional parameter of the graph's edge, enables to build the library with or without dynamic edge attributes, Dijkstra's algorithm and 'cost' attribute. It also takes care of dependency tracking. As the next step the Edmonds-Karp maximum flow algorithm was implemented. To test the introduced changes a series of unit tests was written. The original unit test framework was replaced with *googletest*. The final stage consisted of a series of performance tests. These tests compared the efficiency of the SCGL and BGL solutions in the field of CPU utilisation and memory usage.

In the course of the project the author developed a small, fast, and highly configurable software product. The improved graph library can be used in the network problems research, communication systems research and many others.

Acknowledgements

First of all I would like to thank my supervisor, Ireneusz Szcześniak Ph.D., for continuous support, encouragement and motivating me to work. I am also very grateful to Paula Haapanen from Lappeenranta University of Technology in Finland, my English teacher, for pointing out the strengths and weaknesses of my writing and Arto Kaarna Ph.D. for introduction to the world of C programming. And last but not least, this work couldn't have been done without my colleague from a higher year, Patryk Kwiatkowski, the author and developer of the SCGL library.

Contents

1 Introduction **1**

1.1 The purpose of this thesis 1
1.2 Basic information about the SCGL library 2
1.3 Source code and licence issues 3
1.4 Logical parts and structure of the document 4

2 Methods **5**

2.1 Basic tasks 5
2.2 Build tools review 6
2.3 Using CMake tool 7
2.3.1 Basic syntax issues 9
2.3.2 CMake tutorial 9
2.3.3 Configuration with CMake 13
2.3.4 Defining statically compiled attributes in SCGL 14
2.3.5 Generating documentation with CMake 16
2.4 Implementation of the maximum flow algorithm 16
2.4.1 Preliminary notes on flow networks 16
2.4.2 Ford-Fulkerson algorithm 17
2.5 Unit tests 21
2.6 Performance tests 24
2.6.1 Digression: OpenBSD's MALLOC_STATS 27
2.7 Building the SCGL with CMake 27

3 Results **29**

3.1 Dijkstra's algorithm test 29
3.2 Edmonds-Karp's algorithm test 31
3.3 Memory speed test 32
3.4 Memory size test 32

4 Discussion **35**

5 Conclusions **37**

6 Recommendations **39**

A CD contents **41**

B Oświadczenie **43**

Bibliography **45**

Index **47**

List of Figures

2.1 The flow chart illustrating the build process with CMake 7
2.2 Hello project file structure 10
2.3 An example of flow network 17
2.5 Pathological example of the algorithm complexity 18
2.4 Ford-Fulkerson algorithm 19
2.6 Dijkstra's algorithm directed test graph 26
2.7 Dijkstra's algorithm undirected test graph 26
2.8 Edmonds-Karp algorithm test graph 27

3.1 Bar charts with Dijkstra's results 30
3.2 Dijkstra's performance with different optimisation 30
3.3 Edmonds-Karp's maximum flow algorithm test results 31
3.4 Memory speed results 32

A.1 Contents of the CD attached to this master's thesis 41

List of Tables

2.1 Dependencies in static edge attributes 16
2.2 Examples of the googletest assertions 22

3.1 Dijkstra's shortest path test results 29
3.2 Edmonds-Karp's maximum flow algorithm test results 31
3.3 Memory speed test results 32
3.4 Memory allocation (heap size) test results 33

Listings

2.1 Capacity attribute defined in scgl_config.h file 5
2.2 Building a simple executable in CMake 9
2.3 Building a simple library in CMake 9
2.4 CMake tutorial: CMakeLists.txt file 10
2.5 CMake tutorial: hello_config.h.in file 11
2.6 CMake tutorial: hello_config.h file 11
2.7 CMake tutorial: hello.h file 11
2.8 CMake tutorial: hello.c file 12
2.9 CMake tutorial: test_hello.c file 12
2.10 Example options defined in CMakeLists.txt file 13
2.11 Example of config.h.in file 14
2.12 Generated config.h file 14
2.13 Excerpt from the SCGL CMakeLists.txt 15
2.14 Excerpt from the scgl_config.h file 15
2.15 Example of the timer used for time measurements 25

Chapter 1

Introduction

Graphs are one of the most essential and pervasive elements in modern computer science. Modelling of network systems and traffic between routers, GPS navigation systems, picture segmentation techniques in computer graphics, and even searching of shortest routes in Google Maps®—at the basis of all these things lies a fundamental mathematical concept—graphs. There exist many libraries for creating and manipulating graphs: the Boost::Graph Library (BGL) written in C++ [25], the LEMON graph library and *igraph*—an extensive graph library written in C language [5], to mention only the most popular and commonly used. The following chapter outlines this master's thesis goals and introduces some background information about SCGL library and also presents the structure of this document.

1.1 The purpose of this thesis

The purpose of the research and development work proposed in this thesis is to add a configuration possibility to the *Simple C Graph Library*. The SCGL library was written in the elementary form by Patryk Kwiatkowski for his master thesis at Czestochowa University of Technology and presented to the Faculty of Mechanical Engineering and Information Technology. The SCGL library was implemented without a configuration possibility. The primary objective of this work is to implement this feature to enable configuration of the library sources before the build process. The configuration will be performed with a dedicated build software and mainly involves setting the graph's edge attributes at the time of compilation. In the first version of this software edge attributes were added dynamically by using special functions. The goal is to move this process to the compilation stage and to enable the addition of static attributes and configure other options. Statically compiled attributes will hopefully provide a better performance and shorter execution times. One should observe an improvement in the library efficiency after the changes are implemented. Achieved results should be then compared to the other graph libraries. Effects of the work will be measured by a series of tests. First of all—unit tests, to check integration and correctness of every module from the library. After that one can measure the performance and compare the results to other graph libraries, for instance *Boost::Graph* or *igraph*. This will allow to answer the question,

whether and how the SCGL performance has changed after introduced optimisations. Detailed information on testing is provided in chapter 2. In order to carry out research tasks proposed in this thesis and answer the research questions, this study proposes both—design science and natural science approach[19]. To enhance software quality one needs to follow the methodology used in software engineering. But, simultaneously, answering the research question requires an experimental approach and hence requires methods that come from the natural science.

The next task is to implement the read and write feature in the SCGL library. This should provide the user with the ability to load and save created graphs into a file. The most challenging and tricky part here is that the user should be able to load/save graphs from the libraries with different kind of configuration. For instance, the programmer could have compiled the SCGL library with the “capacity” edge attribute enabled and then saved such composed graph to a file. But next time he/she recompiled the library with the “capacity” attribute disabled. It should be possible to load a saved graph into this version of library without errors.

A subsequent stage is the implementation of an algorithm that uses one of the newly defined attributes, e.g. “capacity”. In this case it should be one of the maximum flow algorithms, such as Ford-Fulkerson, or its improved version by Edmonds-Karp. The implemented algorithm will also allow to test the new statically defined attributes. The final stage consists of a series of unit and performance tests.

1.2 Basic information about the SCGL library

Graphs applications in computer science has been studied extensively in recent years. One of the studies that refers to the outlined topic is the work of Kwiatkowski[17]. The author has developed a *Simple C Graph Library* (SCGL) and also carried out a series of performance tests, comparing his work to the *igraph* and BGL libraries. The results were published in his master thesis. This work is based on and continues the SCGL project.

The SCGL library is an open-source graph library written in the C language. The library can be used to create directed and undirected graphs and to perform operations on graphs. At the moment it implements the Dijkstra’s algorithm for searching the shortest path between vertices. It is worth to mention a few especially interesting and non-trivial features of the SCGL library: it uses the Linux Kernel Lists mechanism to handle lists (lists of vertices and edges in particular), it uses a statically compiled variable type for the edge cost and is small (does not require any additional libraries), flexible and fast graph library. Also it is distributed together with unit tests provided by *DejaGNU* platform.

However, the SCGL library implements only one graph algorithm (Dijkstra’s shortest path). Moreover, in the current version less attention has been paid to configuration scripts and other important utilities like writing graphs to a file and reading from a file. Therefore there is still a space for improvement in this software, for instance by

implementing different kinds of algorithms like maximum flow algorithm. This issues will be dealt with in detail in the next parts.

As mentioned before, the SCGL library uses *Linux Kernel Lists* in implementation. It has five such lists: (i) graph edges, (ii) graph vertices, (iii) edges entering vertex, (iv) edges exiting vertex and (v) edge attributes. This implementation reminds a normal adjacency list, but it differs from the traditional solution in many ways. The normal linked lists usually contains a data field in the list element structure. The *Linux Kernel List* behaves as if the list itself was placed inside of interlinked objects. More information on this topic is provided in the referred work[17, p. 23].

The SCGL is implemented in accordance with best programming practises and naming conventions. It declares its own name space by using the `scgl_` prefix in every library function. The SCGL library is implemented in C language. Since it is a procedural language, it does not support object-oriented programming[14]. It has no polymorphism, inheritance, sub-typing or dynamic dispatch. However, there are some common practises that allow to use some of the object programming features and the author of the library designed the software as if it was build from classes. One can for example build a synonym of the constructor and destructor or hide the structure (class) from the view to discourage a programmer from straight modifications of the structure. There are five main structures or classes in the SCGL library: (i) SCGL::Graph—main structure that stores lists of vertices and edges, (ii) SCGL::Edge—struct for graph edges, it keeps an information on the neighbourhood, references to preceding and following vertices, statically compiled cost (weight) field and dynamic lists of attributes, (iii) SCGL::Vertex—describes vertex, (iv) SCGL::Attr—holds the information of attributes for edges, (v) and SCGL::Algorithms—the implementation of the library algorithms.

The graph issues itself are not the subject of this study. Only the basic terminology is provided. Interested reader should refer to the corresponding literature[4, 6, 12, 26] for more information. Research proposed in this thesis is a continuation of the cited work[17]. It concentrates on developing new qualities in a software product, that can be hopefully applicable in many fields, for instance, in communication system research and many others.

1.3 Source code and licence issues

The SCGL library was created as an open-source software and it is licenced under the terms of GPLv2 (GNU Public Licence version 2). The licence requires that any derivative project must be released under the same licence. The author created a derivative project called “o-graph-ml”, i.e. Open Graph Modelling Library. This project is stored in the git repository [1] on `http://github.com`[2].

[1] Git is a popular distributed control version system, similar to subversion.

[2] `https://github.com/ornithion/o-graph-ml/`

The file and directory structure of the original project has been preserved. All header files and definitions are kept in the `include/` directory and the source files in the `src/` directory. In the main project directory the `CMakeLists.txt` file is placed—this is a main configuration file for CMake generator. A special `build/` directory for out-of-source build was added—it contains all files generated by CMake, *Makefile* and also the built library file, executable and tests. Also the licence file and GPLv2 text was attached to the project. To avoid the need to install an additional software, a directory with the sources of "googletest" was added to the project.

1.4 Logical parts and structure of the document

The first part of this master's thesis describes the background of the study, refers to the previous work in the field and discusses the technologies used in the development. The next part deals with the process of the configuration development and presents a CMake tool and rules of its usage. This part also describes techniques an tools used in the configuration together with common problems encountered in this process and draws possible solutions. The third part presents results of the author's work—it describes the results of performance tests. In the discussion section the result are compared to the previous version of the software and to the BGL graph library. The discussion part also summarises gained results. The conclusion section outlines the main achievements and makes a brief summary. The last section—recommendations—suggests some directions for the future research and development of the project.

Chapter 2

Methods

2.1 Basic tasks

The core task of this work is to implement the configuration of the SCGL library. As indicated in the introductory notes, the primary objective of the implementation of this configuration is to enable the addition of static edge attributes . In other words to determine the attribute type in the process of the library compilation. At the moment this is generally done dynamically (with the exception indicated below) by calling respective functions. The new solution should significantly increase the speed. The first version of the software implemented only one statically compiled edge attribute—"cost" and did not allow the users to freely configure other options. Static attributes are proven to be very fast in comparison to attributes dynamically added when the program is executed.

This goal can be easily illustrated in the code snippet. The library should have a configuration file, that allows to include proper type definitions, depending on defined preprocessor macros. In the case of "cost" attribute, together with type, we must also determine the printing format, as well as minimal and maximal values. Listing 2.1 presents a sample fragment of code from configuration file, with a "capacity" attribute definition. "Capacity" is declared as *int* type.

Listing 2.1: Capacity attribute defined in scgl_config.h file

```
1 #ifndef SGCL_CONFIG_H
2 #define SCGL_CONFIG_H
3
4 #define USE_EDGE_CAPACITY
5
6 #ifdef USE_EDGE_CAPACITY
7     typedef capacity_type_t int;
8 #endif
9
10 #endif
```

The referred master's thesis[17] presented some promising results of performance tests. A series of test measured (i) the usage of RAM during the directed and undirected graphs

creation, (ii) the CPU time elapsed in creating and destroying graphs and, (iii) the time required to find the shortest paths by Dijkstra's algorithm [17, pp. 48-52]. In the first test SCGL used less memory allocations than *igraph*, and not much more than BGL. The second test showed that SCGL is very fast, especially when creating directed graphs. Achieved times are better than in *igraph* case, but worse than in BGL. The Dijkstra's algorithm test showed, that SCGL time performance is the same as in BGL case. The detailed results are presented in the Kwiatkowski's thesis[17], interested reader should refer to this work.

However, these results are still unsatisfactory, SCGL seems to be inefficient especially in the case of undirected graphs. One can notice, that there is a space for improvements, and using configuration to statically set edge attributes will hopefully optimise the performance and allow to get better results. However, the complexity of the algorithms and thus performance comes from the algorithms itself[16, 24]. The author will attempt to take advantage of best programming techniques and knowledge to optimise the necessary features and to tune the library performance if possible. The programming practises described in [13–15] appeared to be invaluably helpful in realizing this project tasks. Also a methods of code refactorin and software development described in the Fowler *et al.* [8] and Martin [20] have had a major influence on the project.

After the configuration unit tests will be carried out. This procedure should lead to an assessment of whether a new configuration and applied changes did not introduced any bugs. In the next step a series of performance tests will be run, which will evaluate what impact have these changes had on the performance of the library.

2.2 Build tools review

To simplify and unify building tasks, the author intended to use a dedicated build software. Originally SCGL was built using a popular and widely known *make* tool. Nowadays many modern tools has been developed to facilitate the building process[7]. From the wide spectrum of software, after initial review, three tools were taken into account: *KConfig*[1], *Scons*[2] and CMake[3]. *Kconfig* is a configuration tool embedded in the Linux kernel. Since the SCGL author himself suggested in his thesis this software, this tool was the starting point. Unfortunately, among its inevitable pros, such as a nice GUI (*Graphical User Interface*) available and powerful possibilities, it has also one serious disadvantage. Because it is embedded, it must be distributed together with the Linux kernel. In the case of a small library as SCGL it is not desirable to unnecessarily increase the sources size. There exist at least two other similar but standalone tools extracted from the Linux kernel: *bst-kconfig*[4] and *kconfig-frontends*[5], but these are not commonly

[1] See the project documentation website for the kconfig language description and more detailed information: `https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt`

[2] Project homepage: `http://scons.org`

[3] The project documentation: `http://cmake.org/cmake/help/documentation.html`

[4] Information retrieved from: `https://www4.cs.fau.de/trac/bst-kconfig/wiki`

[5] Project homepage: `http://ymorin.is-a-geek.org/projects/kconfig-frontends`

used and actively maintained and developed tools and using such tools could cause problems in the project.

The second choice was *Scons*—a Python based tool, which can create configuration files. It uses Python programming language and it is easy in management. In this case a GUI interface for configuration is not available, and configuration would require editing of the source files and this was something that the author wanted to avoid.

The third choice was CMake. This tool is widely known and commonly used among software developers and producers. The name is an abbreviation from **C**ross-platform **Make**. Originally it was inspired by *pcmaker* and *autotools*. It is open-source, multi-platform, it has its own programming language and also allows to create configuration files. It acts as a build system generator, this means, it generates build file for native platform. Finally CMake was chosen, recognised as the most appropriate candidate. Similarly to *KConfig* it can use GUI for configuration phase. For GUI it uses *ncurses* (Unix) or *Qt* library (Unix, Windows®) libraries. As an outcome it produces a build file for the native build environment.

2.3 Using CMake tool

CMake is a cross-platform, open-source build system[11]. It is designed—and this is the most important thing from the programmers point of view—to use together with the native build environments. Particularly in the case of Unix-like systems one can say that it is a layer above the *make* program. Figure 2.1 demonstrates an example build process:

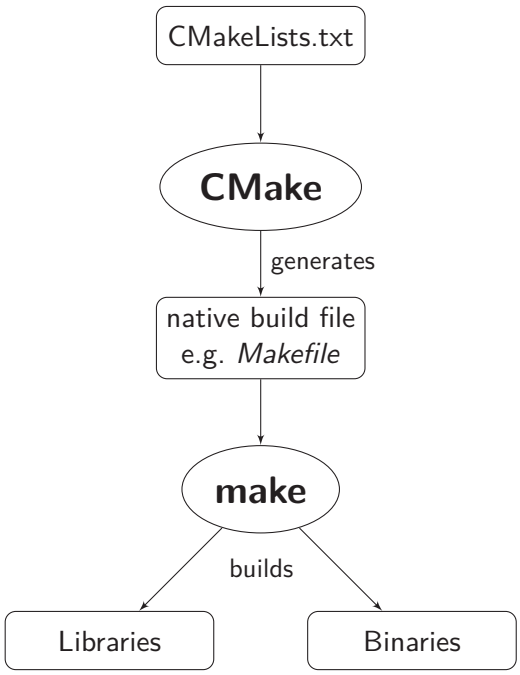


Figure 2.1: The flow chart illustrating the build process with CMake

CMake, as mentioned in the previous section, can generate native build environments. In the case of Unix-like systems it is the *Makefile*, in Microsoft Windows® systems it

can be an IDE (*Integrated Development Environment*) project (e.g. Visual Studio), and on Apple's Mac OS it is the Xcode. This tool is highly flexible, the language syntax is intuitive and supports regular expressions. CMake supports so called "out-of-source" builds, which is very helpful. Also is has integrated CTest utility for testing and CPack for creating packages.

"Out-of-source" build is a very useful feature. It allows to separate all the produced files and binaries from the source files and to keep order in the source tree. To perform this kind of build first one needs to create a "build" directory at the level where a *CMakeLists.txt* file is placed. In the next step one should call one of the CMake commands from that directory (`cmake, ccmake, cmake-gui`—as a general rule this command takes a path to the source files, hence this dot-dot pointer to the parent directory), and then build the sources with the *make* tool:

```
$ mkdir build
$ cd build
$ ccmake ..
$ make
```

Whereas this type of build is recommended, one can choose the normal build, called accordingly "in-source". This procedure is generating all the auxiliary files and binaries in the same directory where the sources are, so it leads to a (unnecessary) mess in the source directories.

CMake can be run in one of three modes: simple mode invoked by typing "cmake" configures the sources with the default options and generates a `Makefile`; "ccmake" (it stands for *curses cmake*) runs a GUI environment in the terminal and allows to freely customise the configure script; "cmake-gui" does the same, but in the windows system. Of course CMake allows to configure most of the options directly from the command-line by passing parameters (`-D` flag).

Configuration is made by creating `CMakeLists.txt` file. Basically these files are placed in each directory of the project. This is the only one file that the developer should really care for. There is no need for editing any other files and one should never change the the content of the files generated by CMake. `CMakeLists.txt` is a simple text file that contains parameters for build. It is written in the CMake language. Usually one needs to describe at least the project name, CMake version and set the input files. If we want to provide the user with the possibility of setting custom definitions, CMake can generate a special *config.h* file, one only need to create an input file for CMake. It can be named anyhow, as long as this name does not conflict with other file names. The good practice is to use either *config.h.in* or *config.h.cmake* name. Any options, that the programmer would like to include in the generated file, must be described in the `CMakeList.txt` file.

2.3.1 Basic syntax issues

CMake provides its own commands and syntax. It is very easy to follow and programmers should not have any problems with understanding and using it. CMake supports uppercase, lowercase and mixed commands. A few of the most commonly used examples are given below:

- `# Comments start with hash`
- Commands syntax: `command(arg1 arg2 ...)`
- Variables: `${VAR1}`
- Conditional statements:
  - `if() ... else()/elseif() ... endif()`
  - `while() ... endwhile()`
  - `foreach() ... endforeach()`

CMake can also use regular expressions, can be used for finding files, libraries, programs and packages. The detailed documentation gives a plenty of examples and deals with nearly every aspect of the tool. It also provides the answers for many questions, so interested readers are referred to it.

2.3.2 CMake tutorial

To get a good understanding of how the CMake works, let us present a minimal working example or so called tutorial[6]. It illustrates the basic rules and mechanisms of building projects with CMake. Basic CMake usage could not be simpler. Building a very trivial executable requires only two lines of code in the `CMakeLists.txt` file. Actually, the first line is not even mandatory.

Listing 2.2: Building a simple executable in CMake

```
1 project(hello C)
2 add_executable(hello hello.c)
```

In the same way one can build a very simple library:

Listing 2.3: Building a simple library in CMake

```
1 project(mylib C)
2 set(LIBSRC mylib.c)
3 # If there is more source files, the procedure is equally simple:
4 # file(GLOB LIBSRC *.c)
5 add_library(mylib STATIC ${LIBSRC})
```

Let us go into more details. The example presented below covers three fundamental aspects of CMake usage: (i) building static libraries, (ii) building executables, (iii) and definition of the configuration options. First one should start with preparing a proper file

[6]Example is based on information found in the CMake documentation; some useful notes were also found in: Jan Engels, "Introduction to CMake"—available as a PDF presentation: `http://ilcsoft.desy.de/portal/e279/e346/infoboxContent560/CMake_Tutorial.pdf`

and directory structure. The whole example can be found in the compact disc attached to this thesis (`cmake_tutorial.tar.gz`). Let us create a simple project named "hello_lib". At the top of the project directory we create a *CMakeLists.txt* file, as in Listing 2.4.

```
hello_lib
  |---CMakeLists.txt
  |---include/
  |    |---hello_config.h.in
  |    |---hello.h
  |---src/
  |    |---hello.c
  |---test/
       |---test_hello.c
```

Figure 2.2: Hello project file structure

The `CMakeLists.txt` file contains settings required to build our example. The code is self explanatory, first we create a project and we call it `hello`. Next we set some variables, that are used later in the testing application. We set the include directory path in order to `CMake` could find all the header files. Finally, we build the `hello` library and then the executable `test_hello`, and link it with the library. Our library does nothing specially interesting—it simply prints out a short welcome message.

Listing 2.4: CMake tutorial: CMakeLists.txt file

```
1 # Source dir is ${project_source_dir}
2 # Binary dir is ${project_binary_dir}
3 # Set the cmake backward compatibility level
4 cmake_minimum_required(version 2.8)
5
6 # Project 'hello' is written in C language
7 project(hello C)
8
9 #set variables and options
10 set(hello_VERSION_MAJOR 1)
11 set(hello_VERSION_MINOR 0)
12 option(STR_LENGTH "Print out the length of the input string?" OFF)
13
14 # Add include dir to find hello_config.h
15 include_directories(${PROJECT_SOURCE_DIR}/include)
16
17 # build the 'hello' library
18 add_library{hello STATIC ${PROJECT_SOURCE_DIR}/src/hello.c}
19
20 # build the executable file
21 add_executable(test_hello ${PROJECT_SOURCE_DIR}/test/test_hello.c)
22
23 # and link it with the library
24 target_link_libraries{test_hello hello}
25
26 # add configuration files path
27 configure_file (
```

```
28     ${PROJECT_SOURCE_DIR}/include/hello_config.h.in
29     ${PROJECT_BINARY_DIR}/include/hello_config.h
30 )
```

The hello_config.h.in file contains information about the definitions, that will be set in the hello_config.h file. In our example this is simply the project version number. The expression embraced by a "@" signs is replaced with the value set in the CMakeLists.txt file. Next comes the #cmakedefine line—after configuration process this will be replaced with the #define or /* #undef */ statement accordingly, depending on how the values are set in the configuration process. By default this is set to OFF. This file is required by CMake to generate a hello_config.h file, which can be later included in the source files in order to read the configuration.

Listing 2.5: CMake tutorial: hello_config.h.in file

```
1 #define hello_VERSION_MAJOR @hello_VERSION_MAJOR@
2 #define hello_VERSION_MINOR @hello_VERSION_MINOR@
3 #cmakedefine STR_LENGTH
```

After running cmake command, we get the hello_config.h file, that contains information about the software version and defined variables. The values of @Hello_VERSION_MAJOR@ and @Hello_VERSION_MINOR@ are replaced with the values from CMakeLists.txt file. Of course this is only very basic and simple example, but is characterises a process of creating configuration files. It consists of a set of definitions. We go in more details of this process in the next section.

Listing 2.6: CMake tutorial: hello_config.h file

```
1 #define hello_VERSION_MAJOR 1
2 #define hello_VERSION_MINOR 0
3 /* #undef STR_LENGTH */
```

The source code of the project is listed below. The test_hello application takes name as an argument, displays a short message with the version of the project and a welcome message. If the STR_LENGTH is defined, it also calculates the number of characters in the given string.

Listing 2.7: CMake tutorial: hello.h file

```
1 #ifndef HELLO_H
2 #define HELLO_H
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include "hello_config.h"
7
8 int hello_msg(char *name);
9
10 #endif
```

The source file of our library:

Listing 2.8: CMake tutorial: hello.c file

```
#include "hello.h"

int hello_msg(char *name) {
    fprintf(stdout, "Welcome %s, it's nice to meet you!\n", name);
#ifdef STR_LENGTH
    fprintf(stdout, "Did you know, that your name has: %d letters\n",
        strlen(name));
#endif
}
```

And the testing application:

Listing 2.9: CMake tutorial: test_hello.c file

```
#include "hello.h"

int main(int argc, char **argv) {
    fprintf(stdout,"This is %s software, Version %d.%d\n", argv[0],
        hello_VERSION_MAJOR, hello_VERSION_MINOR);

    if(argc < 2) {
        fprintf(stdout,"Usage: %s 'your name'\n", argv[0]);
        return 1;
    }

    hello_msg(argv[1]);
    return 0;
}
```

After these short explanations, we can continue to build our project. In the main project directory we create the `build` directory and run `cmake` command. A short information about compiler is displayed and the configuration process is done. This is done automatically, so the project is built with the default settings. If we want to change those settings, we should run either `ccmake` or `cmake -i` instead. In the next step we call `make` and build our application. The binaries and library file are placed in the `build` directory. This location can be altered and CMake can put these files in `lib` and `bin` directories to keep things clean and ordered. The whole process should present itself similar to this:

```
$ mkdir build
$ cd build
$ cmake ../
-- The C compiler identification is GNU 4.2.1
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Configuring done
```

```
-- Generating done
-- Build files have been written to: /home/user/projects/hello/build
$ make
[ 50%] Building C object CMakeFiles/hello.dir/src/hello.c.o
Linking C static library libhello.a
[ 50%] Built target hello
Scanning dependencies of target test_hello
[100%] Building C object CMakeFiles/test_hello.dir/test/test_hello.c.o
Linking C executable test_hello
[100%] Built target test_hello
```

Perspectives of the CMake usage are multiple and not only limited to building executable files. It can be used to build static and shared libraries, projects with many dependencies, it supports unit tests, it can install programs and build packages. Installation of built applications is also possible. With CMake the programmer can also generate a documentation of the project (e.g. with *Doxygen*).

2.3.3 Configuration with CMake

The basic example above presents how to use configuration in CMake in basic form. The rules of creating `CMakeLists.txt` files are simple and the same is true for configuration files. Full description can be found in the CMake documentation. This section concentrates on explaining in more details the configuration process and rules of writing the input files for CMake. General convention for creating entries in configuration files is presented in Listing 2.10. These examples cover also a dependency tracking problem and explain how it can be solved with CMake.

Listing 2.10: Example options defined in CMakeLists.txt file

```
1 # option(VARIABLE "Description" Initial state)
2 option(USE_VAR1 "Enable VAR1 option" ON)
3 option(USE_VAR2 "Enable VAR2 option" OFF)
4
5 # set VAR3 value to 1024
6 set(VAR3 1024)
7
8 # or you can set option this way -- it cannot be changed by the user
9 set(USE_VAR4 ON CACHE BOOL "Build with VAR4 support" FORCE)
10
11 # cmake dependent option useful for dependency tracking: USE_VAR5
12 # only if USE_VAR1 is ON and USE_VAR2 is set to OFF.
13 option(USE_VAR5 "Use VAR5 ON" ON
14         "USE VAR1;NOT USE_VAR2" OFF)
```

CMake requires also the `config.h.in` input file to process defined options. In the `config.h.in` file the programmer writes a code, that depending on options marked in the `CMakeLists.txt` file, either enables or disables appropriate definitions.

Listing 2.11: Example of config.h.in file

```
#ifndef CONFIG_H
#define CONFIG_H

#cmakedefine USE_VAR1
#cmakedefine USE_VAR2
#define VAR3 @VAR3@
#define USE_VAR4 @VAR4@
#cmakedefine USE_VAR5

#endif
```

However, CMake is very flexible tool and user is not restricted to using it only one way. One can also pass configuration options from command line. For instance, we can use `-D` parameter to control configured options:

```
$ cmake ../ -DUSE_VAR1=ON
```

After processing the input file from Listing 2.11, CMake produces this code:

Listing 2.12: Generated config.h file

```
#ifndef CONFIG_H
#define CONFIG_H

#define USE_VAR1
/* #undef USE_VAR2 */
#define VAR3 1024
#define USE_VAR4
#define USE_VAR5

#endif
```

This file should be included in every source file, is going to use the defined variables. In the source files we can now use the preprocessor statements, for instance `#ifdef ...#endif` blocks, to manage the program flow and logic.

2.3.4 Defining statically compiled attributes in SCGL

One of the tasks of the SCGL configuration was to provide the programmer with the possibility to use statically compiled edge attributes only and, for instance, completely turn off the dynamic attributes. This can be done at the compilation level. In the case of statically compiled edge attributes, one can use one of two methods: `add_definitions` function or set necessary variables and output them to `config.h` using CMake language. The former method is more flexible, and makes more sense, when the programmer wants to define many static attributes. Let us take a look at the example of 'cost' edge parameter presented in Listing 2.13. First we set an option `USE_EDGE_COST` that will be defined by user. It is set to `ON` by default. If this is defined we set the type of the cost

parameter. It can be set to one of ten basic variable types, the whole list is presented in description. After setting the type, `cost_fmt`, `cost_max` and `cost_min` are set accordingly.

Listing 2.13: Excerpt from the SCGL CMakeLists.txt

```
1  option(USE_EDGE_COST "Use edge 'cost' attribute" ON)
2
3  if(USE_EDGE_COST)
4      set(EDGE_COST_TYPE "${EDGE_COST_TYPE}" CACHE STRING "Set type of edge
           attribute [short|unsigned short|int|unsigned int|long|unsigned
          long|long long|float|double|long double]")
5  endif()
6
7  if(EDGE_COST_TYPE MATCHES "short")
8      add_definitions(-Dcost_fmt=\"%hd\" -Dcost_max=SHRT_MAX -Dcost_min=
          SHRT_MIN)
9  elseif(EDGE_COST_TYPE MATCHES "unsigned short")
10     add_definitions(-Dcost_fmt=\"%hd\" -Dcost_max=USHRT_MAX -Dcost_min=
          USHRT_MIN)
11
12 . . . truncated for brevity . . .
13
14 else()
15     set(EDGE_COST_TYPE "unsigned int")
16     add_definitions(-Dcost_fmt=\"%d\" -Dcost_max=UINT_MAX -Dcost_min=
          UINT_MIN)
17 endif()
18
19 cmake_dependent_option(USE_DIJKSTRA_ALGORITHM "Build with Dijkstra
       algorithm support" OFF
20     "USE_EDGE_COST" ON)
```

The example above presents an interesting CMake feature called “cmake dependent option”. Using this setting, we can define options that depend on the other options. In this particular case `USE_DIJKSTRA_ALGORITHM` depends on `USE_EDGE_COST`. First it checks, whether this option is set to `ON`. If it is, then `USE_DIJKSTRA_ALGORITHM` is presented to the user and default value is `OFF`. In the same way one can define other dependencies.

If the `USE_EDGE_COST` option is set to `ON`, the user should define the type of “cost” parameter. This is left to the user, however, leaving this option empty is allowed. In this case it defaults to `unsigned int` (see lines 15-16 of the example above).

Listing 2.14: Excerpt from the scgl_config.h file

```
1 #define USE_EDGE_COST
2 typedef unsigned int cost_type_t
3 /* #undef USE_DIJKSTRA_ALGORITHM */
```

The other edge parameters are set in the same way. The values of `{parameter}_max` and `{parameter}_min` are defined in the `limits.h` file in the operating system. Dependencies of these values are presented in Table 2.1 (cf. [17, p. 32]).

Table 2.1: Dependencies between edge parameter type, printing format, maximal and minimal values (parameter = {cost, capacity}).

parameter_type	parameter_fmt	parameter_max	parameter_min
short	%hd	SHRT_MAX	SHRT_MIN
unsigned short	%hd	USHRT_MAX	USHRT_MIN
int	%d	INT_MAX	INT_MIN
unsigned int	%d	UINT_MAX	UINT_MIN
long	%ld	LONG_MAX	LONG_MIN
unsigned long	%ld	ULONG_MAX	ULONG_MIN
long long	%lld	LLONG_MAX	LLONG_MIN
unsigned long long	%lld	ULLONG_MAX	ULLONG_MIN
float	%f	FLT_MAX	FLT_MIN
double	%f	DBL_MAX	DBL_MIN
long double	%Lf	LDBL_MAX	LDBL_MIN

2.3.5 Generating documentation with CMake

Originally the documentation of the SCGL project was generated by Doxygen. Doxygen is a popular source code documentation tool. The author decided to keep the same facility, first and foremost because CMake supports it very well. The main intention was to supply the solution that allows to completely manage the project with a single tool. To build the documentation with CMake, first we must set the `BUILD_DOCUMENTATION` option to `ON`. After running CMake the documentation is build by typing `make Docs`. All the files are placed in the `build/doc/latex` directory.

2.4 Implementation of the maximum flow algorithm

In this section the author describes basic information on the network flows. To understand the algorithms and issues concerned with this part of graph theory, one needs to introduce some basic terminology. Theory in this chapter is based on information found in Ahuja *et al.* and Cormen *et al.* works[2, 4]. Also other studies were found very helpful [6, 12, 23, 27]. The interested reader can find out more on this topic in the referred books.

2.4.1 Preliminary notes on flow networks

We can look at graphs as a networks, in which some kind of medium flows. It can be a water, electricity, data, network packages; it can be a flights, cruises, trains connections and road traffic, and even a blood circulation, ... and many more. One can observe, that the maximum flow problem is closely connected to the shortest path problem. Since the latter was discussed in the Kwiatkowski's work, the author intends to concentrate only on the first problem. It can be shortened to the very simple statement: one needs to send the flow from source to sink, and not exceed the capacity of any edge[2].

The **flow network** is a connected directed graph (or connected digraph) $G = (V, E)$. In the edges there exists a flow of some medium, towards the direction of the edge, and it must not exceed the **capacity** of the edge. This is the new important parameter that we introduce in this chapter. The capacity function id usually marked as $c(u, v) \geq 0$, where $u, v \in V$.

Another basic terms are: the **source** (s)—a vertex from which the flow starts and the **sink** or **target** (t)—vertex into which the flow is entering. All flows start at source and flow down to the sink.

Conveniently we define f function to describe the flow. For each vertex the sum of entering flows equals to the sum of exiting flows—according to the Kirchhoff's law:

$$\sum_{y \in N(x)} f(x, y) = 0.$$

This function needs to satisfy three requirements: (i) **capacity constraints**, for all $u, v \in V, f(u, v) \leq c(u, v)$ so the flow must not exceed the capacity, (ii) **skew symmetry**, for all $u, v \in V, f(u, v) = -f(v, u)$, this implies, that there exists a difference between the flow and the "shipment" of the medium[4, see p. 646], (iii) **flow conservation**, for all $u \in V\{-s, t\}, \sum_{v \in V} f(u, v) = 0$.

The next important terms we need to introduce in order to understand maximum flow problem are: the **residual capacity** and the **augmenting paths**. Residual capacity is a difference between the flow and capacity (usually integer type): $c_f(u, v) = c(u, v) - f(u, v)$. Augmenting path is a path in residual network, that connects the source with the sink. Basically, to solve a maximum flow problem—as it was said before—we need to find the flow from the source to the sink, respecting the capacity constraints of each edge/channel. Since this problem is easier to understand when presented in the graphical form, an example of the flow network is shown in Figure 2.3.

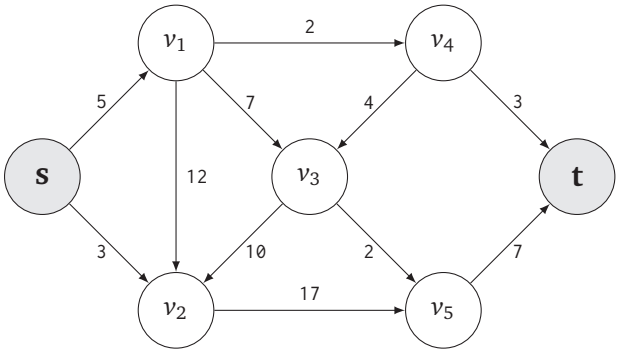


Figure 2.3: An example of flow network. Each edge with *capacity* parameter marked. **s**: source, **t**: sink (target).

2.4.2 Ford-Fulkerson algorithm

From a range of maximum flow methods[3], the author has chosen the Ford-Fulkerson's algorithm. It is widely described and commonly known. Originally it runs in $O(E|f*|)$

time, where $f*$ means the maximum flow found. The first *for* loop runs in $\Theta(E)$ time. As it can be seen, it is strongly dependent on the f value. One can improve this by using Edmonds-Karp modification, then complexity is $O(n^5)$ at maximum. The pseudocode is based on the algorithm presented in chapter 25^{th} of the *Introduction to algorithms* by Cormen *et al.* [4, p. 658].

Algorithm 1 FORD-FULKERSON(G,s,t)

```
1: for each edge (u, v) ∈ E[G] do
2:     f[u, v] ← 0
3:     f[v, u] ← 0
4: while there exists a path p from s to t in the residual network G_f do
5:     c_f(p) ← min {c_f(u, v) : (u, v) is in p}
6:     for each edge (u, v) in p do
7:         f[u, v] ← f[u, v] + c_f(p)
8:         f[v, u] ← –f[u, v]
```

The Ford-Fulkerson algorithm calculates maximum flow. At the beginning all the flows are zeroed. It starts by choosing the augmenting path. This path connects the source with the sink, and all the edges on the way must have zeroed net flows. For better comprehension of the algorithm logic readers should study Figure 2.4 (next page); it presents a sequence of graphs that explain step by step how the maximum flow is calculated. This is one of the most intuitive algorithms, yet it has several disadvantages. For instance, we can consider a typical bad example of the network to see, that in this case the algorithm can run a very long time[2]. Figure 2.5 shows such a network:

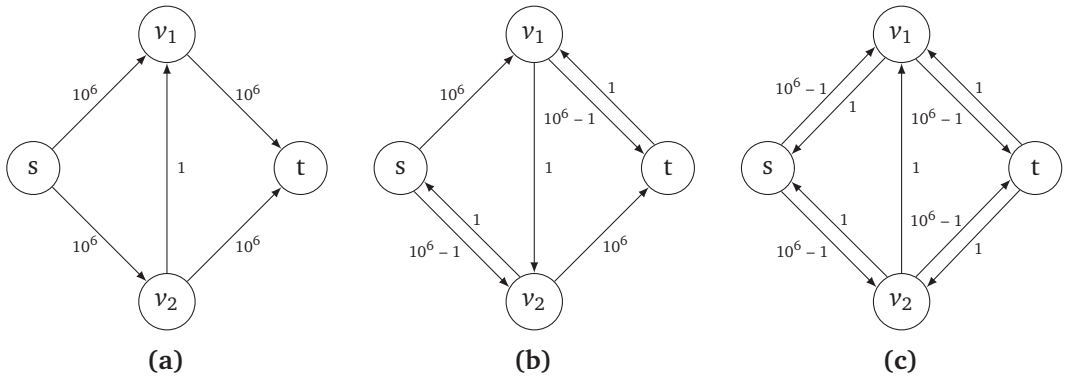


Figure 2.5: Pathological example of Ford-Fulkerson running time: (a) residual network with zero net flow; (b) network flow after finding augmenting path s–v_2–v_1–t; (c) next run of the while loop—augmenting path s–v_1–v_2–t.

As mentioned before, this algorithm can run in $O(E|f*|)$ time, where $f*$ means maximum flow. Usually one cannot optimistically expect a good conditions. It can be shown in a classic pathological example, that the algorithm can run for a very long time. With every step of loop the path is augmented only by 1. It means, that to find the maximum flow in this network, the loop will run 10^6 times.

The version of Edmonds-Karp presumes, that path should be the shortest (in terms of maximal number of jumps and not in terms of cost or weight). Afterwards it finds

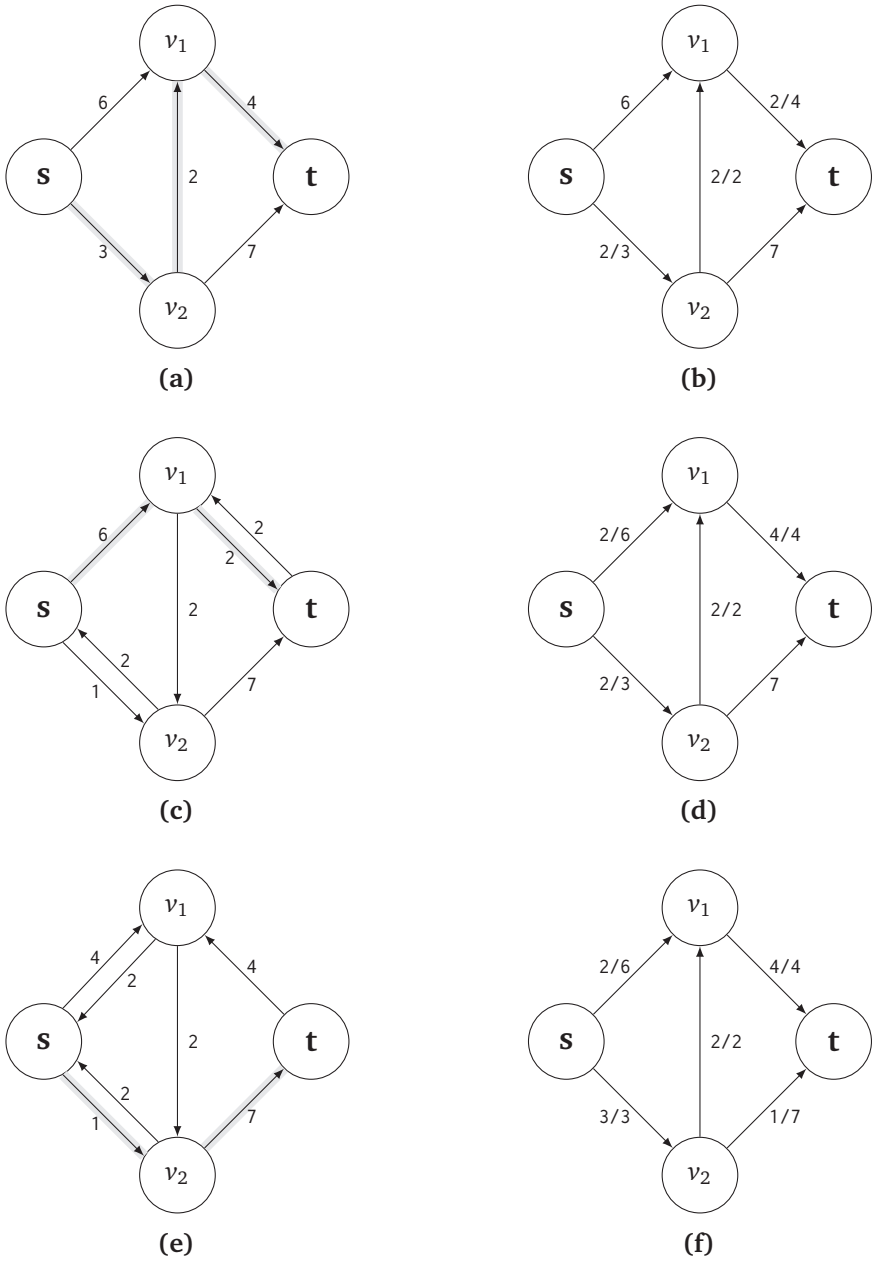


Figure 2.4: Illustration of Ford-Fulkerson augmenting path algorithm: (a) residual network with zero net flow, first augmenting path s–v_2–v_1–t marked with red thick line; (b) network flow after first loop; (c) second augmenting path found s–v_1–t; (d) network flow adjusted accordingly; (e) last augmenting path in this network s–v_2–t; (f) maximum flow found for presented network: $f_{max} = 5$.

the lowest flow, i.e. the lowest capacity, and all the flows in the residual network are decreased with this value. Capacity of such path equals to the lowest capacity of the path edges. Usually it uses breadth-first search algorithm. It can be described by an equation: $c_f(p) = min\{c_f(u,v) \mid (u,v) \in p\}$ (*see line 5 in the pseudocode*). So in the beginning the flow is zero: $|f| = 0$. In the next steps the new flow is enlarged according to this equation: $|f_{new}| = |f_{old}| + c_f(p)$. At the same time the residual capacity decreases. Next steps are repetition of the same procedure, until the maximum flow is found.

Implementation of the maximum flow algorithm proposed in this thesis is based on the optimised version of Ford-Fulkerson from *Algorithms in a Nutshell* by Heineman[10, p. 242]. It uses array to store the information about capacities and flows. This implementation uses a simplified queue. One can consider however, using a queue implemented with Linux Kernel List. A simple version of the queue is included in the source files: `queue.h`. The proposed implementation uses a two-dimensional array to store the values of all the capacities and edges. This is a typical solution optimised for speed, since the time complexity of getting a value from array is $O(1)$. However, this is not a good solution for graphs with large number of vertices and edges. Storing the capacity values of a very complex graph can exceed the available RAM memory. The algorithm is called Ford-Fulkerson, but in fact this is a specialised version of Edmonds-Karp that finds the augmenting paths with breadth-first search algorithm. To get the proper results, edge capacities must be integer type. The algorithm itself presents as follows:

Algorithm 2 FORD-FULKERSON(G,s,t)

```
 1: function COMPUTE(G)
 2:     while find augmenting path in G do
 3:         processPath(path)
 4: function PROCESSPATH(path)
 5:     v = sink
 6:     delta = ∞
 7:     while v ≠ source do
 8:         u = vertex previous to v in path
 9:         if edge(u,v) is forward then
10:             t = (u,v).capacity – (u,v).flow
11:         else
12:             t = (v,u).flow
13:         if t ≤ delta then
14:             delta = t
15:         v = u
16:     v = sink
17:     while v ≠ source do
18:         u = vertex previous to v in path
19:         if edge(u,v) is forward then
20:             (u,v).flow += delta
21:         else
22:             (v,u).flow –= delta
23:         v = u
```

The algorithm finds an augmenting path in given graph using a breadth-first search (BFS). In the beginning all the vertices are marked as not visited (white). The BFS starts in the sink—the vertex is pushed to the queue and marked with the gray colour. Then the algorithm checks all the neighbouring vertices, whether they have enough capacity. Searching finishes when the sink is reached. Visited vertices are marked with the black colour. In parallel, the flow is augmented along the found paths until the maximum flow is found.

2.5 Unit tests

In the original version of the library, the DejaGNU testing tool was used. Since it's not so commonly used and relatively complicated in use, the author decided to change the unit testing framework. At first look there is a big number of tools to choose from. But what it was looked for, it was the tool, that can be easily integrated into library distribution, in order to provide a developer with the testing framework, to check the effects of his/her work. The Google C++ Testing Framework, usually referred in short as *googletest* has been chosen[7]. Although the *googletest* is written in C++and intended for testing a C++code it is still applicable in C applications. With certain limitations it can be (and it is) successfully used in testing the C code. Things to have in mind when testing a code written in C are among others: static initialisation, global variables and hardware access[18].

Unit tests are very important part of every software development process, the same is true also in the case of the SCGL library. The Google C++ Testing Framework makes it easy to write the tests. It also cares, that tests meet basic rules of unit testing, i.e.:

1. Tests should be independent—one test should not interfere with another. This guarantees repeatable results.
2. Tests should be reusable—writing a test should be a single activity: we write a test once, and use it multiple times in the development process.
3. Tests should be portable, it cannot be necessary to rewrite tests to use them in different environments.
4. When a test fails, it should provide as much information as it is possible, to identify the problem.

Googletest provides a nice API (*Application Programming Interface*) for the programmer. For writing the test a set of macros is provided. For single tests we can use `TEST()` macro:

```
TEST(test_case_name, test_name) {
. . .  test body . . .
}
```

For the tests which operate on the same set of data we can use `TEST_F()` macro that provides so called *test fixtures*. In this way a test suite for SCGL was prepared. What we

[7] Project homepage: `http://code.google.com/p/googletest/`

test are the assertions and as a result we can get a success or failure message. There are two main types of the assertions: fatal- and nonfatal assertions (see Table 2.2). Basic tests cases are derived from the main testing class provided in the testing tool.

Table 2.2: Examples of the googletest assertions

Fatal assertion	**Nonfatal assertion**	**Verifies**
`ASSERT_TRUE(condition);`	`EXPECT_TRUE(condition);`	`condition` is true
`ASSERT_FALSE(condition);`	`EXPECT_FALSE(condition);`	`condition` is false
`ASSERT_EQ(val1,val2);`	`EXPECT_EQ(val1,val2);`	`val1==val2`
`ASSERT_NE(val1,val2);`	`EXPECT_NE(val1,val2);`	`val1!=val2`
`ASSERT_STREQ(str1,str2);`	`EXPECT_STREQ(str1,str2);`	strings are equal
`ASSERT_STRNE(str1,str2);`	`EXPECT_STRNE(str1,str2);`	strings not equal

The unit tests for every module/function from the library was written. The test are divided into subclasses. The main class is derived from `::testing` class, and the particular test cases are derived from it. Twenty-five test cases were created in the test suite:

I. Vertex tests
   1. Vertex create/destroy
   2. Vertex get/set ID
   3. Vertex add/get/delete edge
   4. Vertex count edges
   5. Vertex for each edge

II. Edge tests
   1. Edge create/destroy
   2. Edge get/set cost
   3. Edge get/set capacity
   4. Edge add/get/delete vertex
   5. Edge add/get/delete attribute
   6. Edge count attributes
   7. Edge for each attribute
   8. Edge get/set directed/undirected

III. Dynamic attributes tests
   1. Attribute create/destroy
   2. Attribute get/set key
   3. Attribute get/set value

IV. Graph tests
   1. Graph create/destroy
   2. Graph get/set ID
   3. Graph get/get/delete vertex
   4. Graph add/get/delete edge
   5. Graph count vertices/edges
   6. Graph copy

V. Algorithms tests
    1. Dijkstra's algorithm undirected
    2. Dijkstra's algorithm directed
    3. Edmonds-Karp's algorithm test

Basically all the tests use assertions from the googletest. Three of the tests from the test suite required more advanced technique of testing, which was not available in the googletest. To test some cases the author used a sibling project called "Google C++ Mocking Framework"— googlemock[8]. Although we do not use any real mocking in the SCGL project, yet this tool provides very useful functions to test arrays and regular expressions.

In order to build and run the unit tests there is no need to install the googletest. The development version of the SCGL library has the sources of the googletest already included. This method was selected to simplify the testing process and, since the source files of the googletest are small, it does not affect the overall size of the package. The procedure presents as follows: first, in the configuration stage, one should set the `BUILD_WITH_UNIT_TESTS` option to `ON` to compile the proper files into the library. After that one can perform a normal build with the `make` command. In the `bin` directory we will find the generated tests: `ographml_test`. An example output can look like this:

```
$ ./ographml_test
[==========] Running 24 tests from 5 test cases.
[----------] Global test environment set-up.
[----------] 5 tests from VertexFunctionTest
[ RUN      ] VertexFunctionTest.VertexCreateDestroy
[       OK ] VertexFunctionTest.VertexCreateDestroy (0 ms)
[ RUN      ] VertexFunctionTest.VertexGetSetID
[       OK ] VertexFunctionTest.VertexGetSetID (0 ms)
[ RUN      ] VertexFunctionTest.VertexAddGetDelEdge
[       OK ] VertexFunctionTest.VertexAddGetDelEdge (0 ms)
[ RUN      ] VertexFunctionTest.VertexEdgesCount
[       OK ] VertexFunctionTest.VertexEdgesCount (0 ms)
[ RUN      ] VertexFunctionTest.VertexForeachEdge
[       OK ] VertexFunctionTest.VertexForeachEdge (0 ms)
[----------] 5 tests from VertexFunctionTest (0 ms total)
. . .  truncated for brevity . . .
[----------] Global test environment tear-down
[==========] 25 tests from 5 test cases ran. (0 ms total)
[  PASSED  ] 25 tests.
```

If something goes wrong and some of the tests fail, we will get an illustrative and descriptive information about the failing test. Additionally, we can provide a more descriptive output from the test by writing our own comments. It this case an example output can look like this:

[8]The project website: `http://code.google.com/p/googlemock/`

```
[ RUN      ] EdgeFunctionTest.EdgeGetSetCost
Value of: scgl_edge_get_cost(e1)
  Actual: 123
Expected: 321
[  FAILED  ] EdgeFunctionTest.EdgeGetSetCost (1 ms)
. . .  truncated for brevity . . .
[==========] 25 tests from 5 test cases ran. (1 ms total)
[  PASSED  ] 24 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] EdgeFunctionTest.EdgeGetSetCost

1 FAILED TEST
```

The programmer should remember to enable all the configurable options before building the tests. It is important, otherwise the tests will not compile. To test all units, every feature must be set to `ON`. The tests can also be run using `make tests` and `ctest` commands. The detailed description of these commands is provided in the CMake documentation. One should remember, that unit tests are included in the version of the library called "development version". Once all the test are passed and the developer would like to move the project to the production stage (e.g. in the embedded systems), one can remove the directories with googletest, and make the changes in the `CMakeLists.txt` file accordingly. This will decrease the size of the package.

2.6 Performance tests

One of the purposes of this work was to improve the speed, memory usage and performance of the SCGL library. The original library was tested against the Boost::Graph and igraph libraries. The igraph memory allocation techniques are quite different, so the author decided to carry out a series of performance tests only in comparison to the BGLlibrary; test were performed in four areas, each test (with the exception of Edmonds-Karp) has two subtests, for directed and undirected graphs:

1. CPU utilisation during creation and destruction of a graph;
2. Heap size while running above test (and number of allocations);
3. Dijkstra's shortest path algorithm performance time;
4. Edmonds-Karp's maximum flow algorithm performance time;

Test were prepared in the following manner. Each test was performed in three different environments. This was due to problems, that the author encountered while running the tests on 64-bit architecture computer. The difference between the performance times on 32 bit and 64 bit was not so important in the case of the SCGL library. But in the BGL the differences became more noticeable. So the main purpose of running these tests in different architectures was to investigate the cause of such differences. Test were built with `-s -Os -W -Wall` compiler flags.

The testing environments are described below:

1. 64 bit OpenBSD 5.3
   - processor Intel Core 2 Duo CPU L7500 1.600 GHz 798.15 MHz
   - 2005 MB RAM
   - compiler version gcc-4.7.2
2. 64 bit FreeBSD 9.1
   - processor Intel Core 2 CPU 6400 2.13 GHz (2128.06 MHz K8-class)
   - 4096 MB RAM
   - compiler version gcc-4.6.3
3. 32 bit OpenBSD 5.3
   - processor Intel Pentium D 2x2.80 GHz GenuineIntel 686-class
   - 1498 MB RAM
   - compiler version gcc-4.7.2
4. 32 bit Linux Mint 13 Maya (Linux 3.2.0.23 generic i686)
   - processor Intel Pentium M 1.70 GHz
   - 1025 MB RAM
   - compiler version gcc-4.6.3

The algorithms tests were run in a loop of 10 000 iterations, because of small sizes of tested graphs and because it has allowed to get more adequate and reliable results. Every test application has a built-in timer. The system `clock()` (see `man clock`) function was used for the time measurements. According to manual page, this function measures the amount of time that the CPU spends on a process since its invocation. This solution is trying to minimise the overload bias that comes from the application and tools used for measurements (e.g. variable initialisation, settling the environment, `time` command). The code of the timer is presented in Listing 2.15.

Listing 2.15: Example of the timer used for time measurements

```
#include <time.h> /* or <ctime> in .cpp files */
unsigned int i;
clock_t start, stop;

/* Initial time stamp */
start = clock();

/* The function, that running time we want to measure */
for(i=0; i < 10000; ++i) {
    tested_function();
}

/* Final time stamp */
stop = clock();

printf("Elapsed time: %d ms\n", (int)(1000.0 * (stop - start) /
    CLOCKS_PER_SECOND));
```

Graphs used in Dijkstra's test were the same graphs as in the original tests. First picture shows a graph used in tests for directed graphs, the second one—graph used in tests for undirected graphs. In both cases the algorithm searched shortest paths starting from the vertex 0.

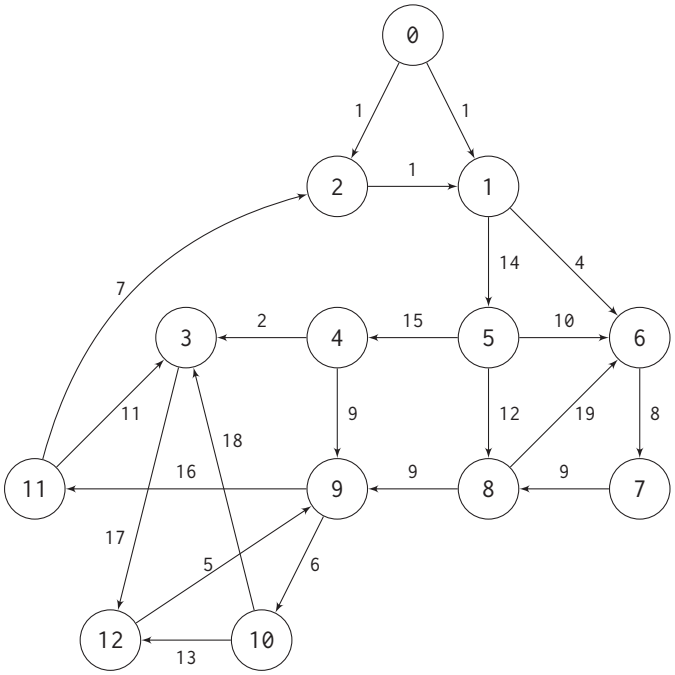


Figure 2.6: Directed graph used for testing Dijkstra's shortest path algorithm

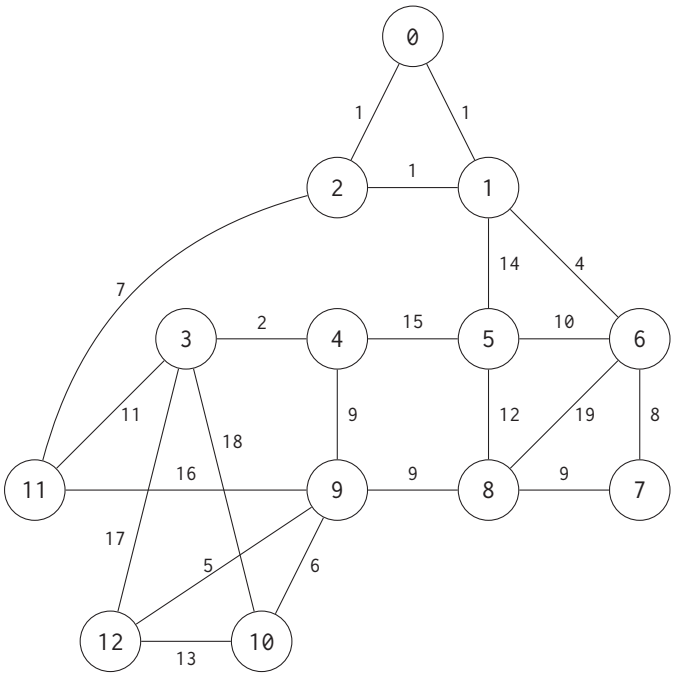


Figure 2.7: Undirected graph used for testing Dijkstra's shortest path algorithm

Edmonds-Karp maximum flow algorithm was tested on the example graph from the Boost::Graph library. The source of the flow network is vertex 0 and 7 is the sink.

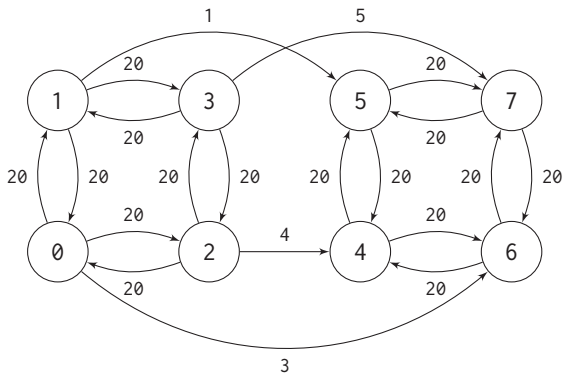


Figure 2.8: Directed graph used for testing Edmonds-Karp's algorithm

2.6.1 Digression: OpenBSD's MALLOC_STATS

Since the OpenBSD does not have any tool similar to *valgrind*[9], it is worth to mention a mechanism, that enables to track memory allocations and detect possible memory leaks. According to `malloc` manual page[21], OpenBSD's implementation considerably differs from other Unix-like systems. Also the system developers confirm that it is very useful in the memory debugging[22]:

> *OpenBSD's malloc is a bit of a different beast compared to most other implementations: it has built-in, always-on randomization and it returns pages to the operating system when it no longer needs them.*

To test the executable in OpenBSD it should be run with `MALLOC_OPTIONS=D` enabled:

```
$ cp /dev/null malloc.out
$ MALLOC_OPTIONS=D ./tested_executable
$ cat malloc.out
```

The `malloc.out` file in the current directory contains a full report about memory allocations and frees and also—if applicable—displays the addresses where the memory is still available (leaks).

2.7 Building the SCGL with CMake

To build the project it is required to have CMake already installed in the system. The next step is getting the sources, either from the `https://github.com/ornithion/o-graph-ml/` or from the CD attached to this thesis. If the sources are archived in the `o-graph-ml.tar.gz` file, first one should unpack the file:

```
$ tar -xfvz o-graph-ml.tar.gz
```

Full description of the file and directory structure is given in the appendix A. The compilation is done in a build directory (out-of-source build), so one must create it first:

[9] Valgrind is a program for debugging and profiling Linux applications. It is commonly used for detecting a memory leaks in programs.

```
$ mkdir build && cd build && ccmake ..
$ make
```

In the `build` directory a `Makefile` has appeared. This file must not be edited, any changes can be done only in the `\CMakeLists.txt` file and corresponding configuration file (`scgl_config.h.in`). Any configurable features that programmer wants to have in the library, must be enabled in the configuration. To change this setting the library must be rebuild. If the default options are fine and there is no need to change that, simply invoke:

```
$ cmake .. && make
```

Using the newly implemented algorithm in the SCGL library is easy. The library must be configured with the `USE_EDGE_CAPACITY` option set to `ON`. Type of the `capacity` variable is determined from the setting of the `EDGE_CAPACITY_TYPE`. It is recommended yet not required to explicitly define this variable type. In case it is not defined, this value is automatically set to `unsigned int`. Also the `USE_EDMONDS_KARP_ALGORITHM` must be set to `ON`. In a similar manner the configuration of Dijkstra's algorithm is done, as it was presented earlier in the chapter.

Chapter 3

Results

First of all one should realise, that real performance tests are beyond the scope of this thesis. Professional tests require a complicated profiling analysis, extended data sets, for example a set of graphs with different vertices number and different complexity. Gathering data from such tests would allow to interpret the results in broader spectrum and lead to the adequate conclusions about time and memory complexity. What we do here is in fact just a basic speed (efficiency) test, performed in different environments, using only one test graph. Performance test were set in four different environments, as described in the previous chapter. This part of the document presents generally only the raw data. Discussion is presented in the next chapter.

3.1 Dijkstra's algorithm test

Table 3.1 shows the results of the Dijkstra's shortest path algorithm test. Tests were run in a loop of 10 000 iterations. Tests measured the CPU utilisation during the Dijkstra's algorithm test.

Table 3.1: Dijkstra's shortest path test results

System	Graph type	Time (ms)	
		SCGL	BGL
OpenBSD 5.3 x86	directed	50	40
	undirected	70	60
OpenBSD 5.3 amd64	directed	100	160
	undirected	140	220
FreeBSD 9.1 amd64	directed	156	85
	undirected	156	64
Linux Mint 13 x86	directed	100	100
	undirected	150	180

Results differ in different environments. As one can see it is very hard to draw consistent conclusions. Test behaviour is completely different in 32 bit and 64 bit systems. Moreover even within the same architecture there is a big variety in the results. On the other hand this seems to be quite natural—different machines have different CPU's caches.

Nonetheless this behaviour is somewhat puzzling. The results of the heap usage throw some more light on this issue (see Table 3.4). In the BGL the number of allocations is doubled.

The same table is presented as the bar charts in Figure 3.1. It is worth to mention, that in the case of Linux system, the results are comparable to the results of the original version of the library. Differences are discussed in the next chapter.

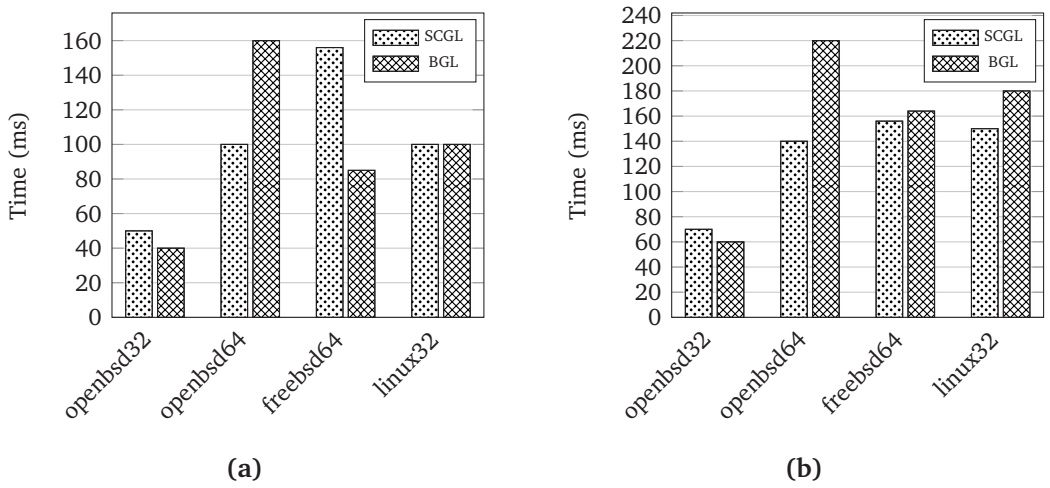


Figure 3.1: Result of Dijkstra's test in (a) **directed** and (b) **undirected** graph

Aside of running Dijkstra's algorithm tests in different environments, the same tests were run on a single 32 bit OpenBSD machine. Test were compiled with four different optimisation levels. Results are shown in Figure 3.2.

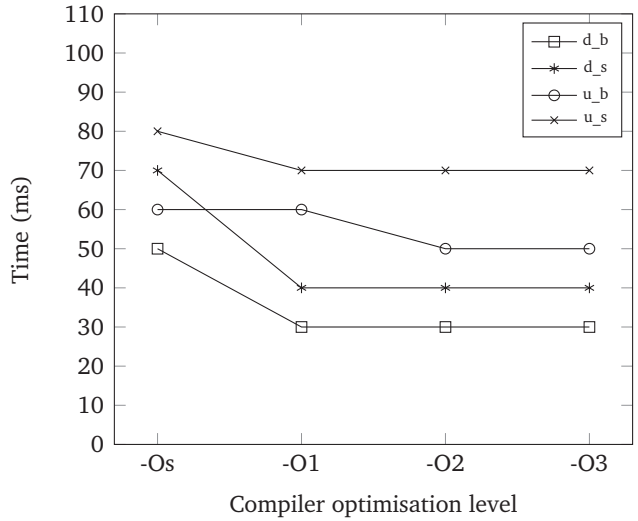


Figure 3.2: Dijkstra's algorithm performance depending on gcc compiler optimisation level. Symbols in legend: **d** directed graph, **u** undirected graph, **s** SCGL, **b** BGL

This experiment led to few interesting observations. One can notice a specific trend here: in the case of the directed graphs both libraries are easy to optimise. When we turn on the level `-Os`—size optimisation—the efficiency is the lowest in both libraries. It increases after the `-O1` level and remains stable in more aggressive optimisation. In

the case of undirected graphs both libraries are less susceptible to optimisation and the efficiency gained in this process is in the range of 10 milliseconds, whereas in directed graphs efficiency can increase by 20 (BGL) to 30 (SCGL) milliseconds.

The SCGL library is less sensitive to optimisation. When no optimisation is done (for instance in profiling with gcc compiler flags set to `-ggdb -pg`) the SCGL library running times are comparable to times measured in the library compiled with optimisation. On the contrary, in the BGL case the same measurements show much longer running times when there is no optimisation. The tests have shown, that this factor is in the range from 5 – 9.

3.2 Edmonds-Karp's algorithm test

Edmonds-Karp's algorithm tests involved finding the maximum flow in tested graph. Tests measured a processor time utilisation during the Edmonds-Karp's algorithm performance. In this case tests were also run in a loop of 10 000 iterations due to small size of tested graph. Correct result (maximum flow = 13) were found faster by BGL. The only exception here is FreeBSD system, maximum flow in this case was found two times faster by the SCGL library. Also in 64 bit OpenBSD results were similar.

Table 3.2: Edmonds-Karp's maximum flow algorithm test results

System	Graph type	Time (ms)	
		SCGL	BGL
OpenBSD 5.3 x86	directed	310	120
OpenBSD 5.3 amd64	directed	810	800
FreeBSD 9.1 amd64	directed	290	630
Linux Mint 13 x86	directed	880	340

As it is clear from the definition of the flow network, a graph must be directed. The tests of Edmonds-Karp's algorithm were conducted using the example digraph from BGL library (see Figure 2.8)

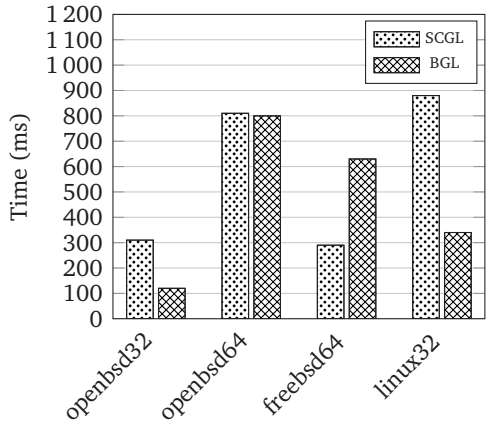


Figure 3.3: Edmonds-Karp's maximum flow algorithm test results

3.3 Memory speed test

This test consisted of creating and destroying a set of 1 001 vertices and 1 000 edges in a graph. Test were run in a loop of 1 000 iterations. A measured factor was a CPU time utilisation during the creation and destruction of graphs. In this case SCGL efficiency is better than BGL in four environments. The greatest range of variation was achieved in FreeBSD system. Results are presented in Table 3.3.

Table 3.3: Memory speed test results

System	Graph type	Time (ms)	
		SCGL	BGL
OpenBSD 5.3 x86	directed	600	620
	undirected	880	1 040
OpenBSD 5.3 amd64	directed	1 040	3 120
	undirected	2 030	3 890
FreeBSD 9.1 amd64	directed	300	700
	undirected	440	950
Linux Mint 13 x86	directed	430	600
	undirected	620	780

Figure 3.4 shows that in undirected graphs measured times were significantly larger. Also here one can notice, that SCGL performance is better than BGL.

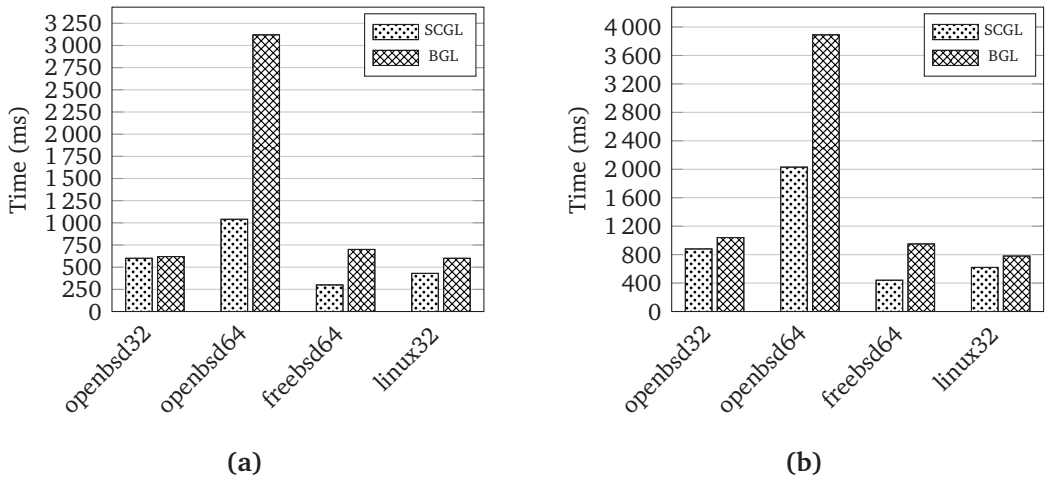


Figure 3.4: Memory speed during the creation and destruction of the graph vertices and edges in: (a) **directed** and (b) **undirected** graph

3.4 Memory size test

This test was similar to the previous one. It consisted of creating and destroying a set of 1 001 vertices and 1 000 edges in a graph, but this was done only once. Test measured heap utilisation during the creation and destruction of a graph, in other words a number of bytes allocated and a number of allocations. The results of heap usage are summarised, i.e. in the parentheses the total number of allocations and frees is given.

Table 3.4: Memory allocation (heap size) test results

System	Graph type	Heap usage in bytes (allocations)			
		SCGL		BGL	
OpenBSD 5.3 x86	directed	112 640	(4 028)	116 736	(4 027)
	undirected	178 176	(6 043)	104 448	(6 034)
OpenBSD 5.3 amd64	directed	217 088	(4 052)	122 880	(10 061)
	undirected	282 624	(6 082)	188 416	(12 121)
FreeBSD 9.1 amd64	directed	152 096	(4 004)	88 519	(10 052)
	undirected	248 096	(6 004)	156 503	(12 052)
Linux Mint 13 x86	directed	80 048	(4 004)	44 756	(4 024)
	undirected	132 048	(6 004)	76 745	(6 024)

An interesting thing here is the number of allocations in BGL library on 64 bit systems. Whereas on 32 bit BGL usually allocates the memory similar number of times, in the 64 bit there is two times more allocations.

Chapter 4

Discussion

Based on the results we can outline the most important conclusions. The SCGL library is faster than the BGL library, especially in creating and destroying graph elements. This results are cross-sectional, because the same results were obtained in four testing environments with a varying range of variation. There is a consistent trend occurring in the results, so we can generalise this conclusion. Also here the results agree with previous work (see [17, p. 50] for your reference). The percentage difference in speed between two tested libraries ranges from 3.2% to 66.6% in directed and from 15.3% to 53.6% in undirected graphs. In both cases the greatest variation is observed in FreeBSD 9.1 system. Smaller differences can be noticed in undirected graphs, perhaps because of the SCGL way of creating undirected edges, which are just siblings (copies) of the original edge.

The things get more complicated in the case of Dijkstra's algorithm. The test behaviour is unclear in different environments. There is no consistency in 32 bit systems and neither is in 64 bit. The author could not determine the reason of this phenomenon. A possible answer to this question might be the difference in size of the variables and pointers in 32 bit and 64 bit architectures. To examine this hypothesis one should use portable type definitions (see more information in [1]).

- `ptrdiff_t`, an integer type, as the name suggests comes from two pointers substraction;
- `size_t`, an unsigned integer, returned by the `sizeof` operator;
- `int32_t, uint32_t` and others: types of a predefined width;
- `intptr_t` and `uintptr_t`, any valid `void*` can be casted to one of these;

Unfortunately the author did not solve this problem because this would require an additional amount of time. As to Edmonds-Karp's solution, generally speaking it performed better in the BGL library in 32 bit systems (61.2%). In 64 bit systems there was a slight difference in favour of BGL (1.23% in OpenBSD), possibly due to measurement error, and a significant difference in favour of the SCGL (53.9% in FreeBSD).

Interested observations can be made from the heap usage results. In the case of BGL library on 64 bit systems the number of memory allocations is twice as big as in 32 bit.

As mentioned before the results are strongly dependent on the optimisation level and in SCGL library increasing optimisation level did not reflect in achieved results as much as it did in BGL. We are talking here about the time complexity of the algorithms. If the performance tests have showed, that proposed solution does not achieve satisfactory results, there are two ways of dealing with this problem. One possible solution is to rethink the data structures and/or algorithms used in the library, another is try to optimise or tune the code (compare chapter 7 in [14]). One caveat here: nowadays compilers do most of the work for us. This is particularly true in the case of the optimisation. If the algorithm and data structures are optimal, it is nearly impossible to beat compilers in optimisation. The good reason to do not try this is that the effects can be sometimes opposite to intentions.

Regardless of the real cause of differences in results of the Dijkstra's algorithm test is is worth to state a fundamental rule on the time and computational complexity—it comes mostly from the algorithm itself. In the case of the Dijkstra's shortest path algorithm both libraries are using the same version of algorithm with Fibonacci's heaps. Moreover SCGL implementation is based on that one from BGL. An interesting challenge would be to implement paralell version of the SCGL library. The real performance time could be shortened by using threads[9].

Chapter 5

Conclusions

The configuration possibility makes the SCGL an easy manageable and highly flexible graph library. It also makes it more portable, because distributed code is independent from the platform. It can be equally easy build in Unix-like environments (Linux and BSD systems, AIX) as well as in Windows® or Apple's Mac OS. The only requirement is a working CMake installation.

All things considered it should be noted, that the major objective of this work has been achieved and milestones indicated in the research plan were successfully completed. As usual in the case of creative projects, there is still much work to be done. When working on the project some new ideas questions has aroused, most important of them are highlighted in the recommendations section.

The main purpose of this work was to improve the SCGL library by adding the configuration possibility. This has been fully achieved. Moreover, some additional tasks has been done: the maximum flow algorithm has been implemented, a new unit test framework has been added and an extended performance test routines has been prepared. With the CMake tool one can make a build in one step. In this step one can customise the library to suit it to ones needs, build the unit tests, build the performance tests and documentation—so might look a set of requirements for the professional software.

The author has learnt new things, especially in the field of graph theory, and seen what are the trends in the software development in this area. Also gained a new practical knowledge, improved the C language programming skills, and reviewed and consolidated fundamentals of computer science. Work on the project was also a good school of software engineering, so the specialization, that the author has chosen as the direction of the future development.

Chapter 6

Recommendations

The SCGL library is still a work in progress. Although it is a complete library already, it has some fundamental features like creating and destroying graphs and its elements and also implements basic algorithms, one must admit, that there is a place for further improvements. Comments made in this section are intended to help in the development of the library.

In few short points listed below the author wants to propose some recommendations and features, that can be implemented in the library in the future.

- Implement more basic graph searching algorithms, like Bellman-Ford or Prim's algorithm—this was already proposed by the original author of the library.
- Implement a better solution for maximum flow finding, consider push-relabel algorithm, or Boykov-Kolmogorov algorithm.
- Modularise basic graph traversing algorithms, like Breadth-First Search and Depth-First Search and put the implementations in separate header files. This step would require rebuilding the implementation of Dijkstra's algorithm.
- Consider enhancing the configuration possibilities, for instance, one can propose more statically defined attributes, introduce attributes defined by the user, etc.
- Consider further modularization in the style of Boost::Graph—only required modules are included in source by including proper header files.

Still one of the most important things to implement is read and write to a file. Perhaps the best solution here would be to implement this feature in one of the commonly used formats like DIMACS[1]. What is the motivation? Reading and writing graphs to a file makes it easier to write programs and much more. In the case of SCGL library it would allow to prepare an extended test routines. It is much easier to create and read a huge graph from a file, so we can test the SCGL with 100, 1 000, 10 000 vertices and get reliable results.

As stated in the discussion section, the causes of the different performance in 64 bit architectures remain unknown. This topic needs more research.

[1] Center for Discrete Mathematics and Theoretical Computer Science `http://dimacs.rutgers.edu/`

Appendix A

CD contents

Compact disc attached to this thesis contains the following items:

```
ographml/
   |---.git/
   |---CMakeLists.txt
   |---doc/
   |    |---CMakeLists.txt
   |    |---Doxygen.in
   |    |---latex/
   |    |     |---manual.pdf
   |    |---thesis/
   |    |     |---thesis.pdf
   |---include/
   |---src/
   |---tests/
   |     |---CMakeLists.txt
   |     |---googletest/
   |     |---perf_tests/
   |     |---unit_tests/
   |---GPLv2.txt
   |---LICENCE
   |---README
```

Figure A.1: Contents of the CD attached to this master's thesis

- the main project directory is called `ographml`;
- `.git` directory holds necessary *git* files and a history of the repository;
- `CMakeLists.txt` file in the main catalogue is a basic CMake file used for configuration and build file generation;
- `doc/` contains project documentation generated by Doxygen (`manual.tex`) and this master's thesis (`thesis.pdf`);
- `include` directory contains the library header files;
- `src` directory contains the library source files;
- `tests` directory includes unit tests and performance tests, it contains also the sources of googletest and googlemock, so installing these tools in the system is not

required;

- `LICENCE` file contains information about this project licence;
- `GPLv2.txt` is the GNU Public Licence, version 2. It is attached according to the requirements of the GNU licenced software;
- `README` file gives some basic information about the project and author's contact data;

Additionally, the project compressed tarball (`ographml.tar.gz`) as well as the CMake tutorial (`cmake_tutorial.tar.gz`) are included.

Appendix B

Oświadczenie

Imię i nazwisko: Marcin Ptak
Numer albumu: 115501
Kierunek: Informatyka
Wydział: WIMiI
Politechnika Częstochowska

Częstochowa, 2013-11-07

Szanowny Pan(i) Dziekan

Pod rygorem odpowiedzialności karnej oświadczam, że złożona przeze mnie praca dyplomowa pt. “Implementacja konfiguracji biblioteki grafowej SCGL” jest moim samodzielnym opracowaniem.

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

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

.....

podpis

*nieodpowiednie skreślić

Bibliography

[1] ADIGA, H. S. Porting Linux applications to 64-bit systems. Retrieved from: http://www.ibm.com/developerworks/library/l-port64/, Aug 2013.

[2] AHUJA, R., MAGNANTI, T., AND ORLIN, J. *Network flows: theory, algorithms, and applications*. Prentice Hall, 1993.

[3] AHUJA, R. K., AND ORLIN, J. B. A fast and simple algorithm for the maximum flow problem. *Operations Research 37*, 5 (1989), 748–759.

[4] CORMEN, T., LEISERSON, C., RIVEST, R., AND STEIN, C. *Introduction To Algorithms*. MIT Press, 2001.

[5] CSÁRDI, G., AND NEPUSZ, T. The igraph software package for complex network research. *InterJournal Complex Systems 1695* (2006).

[6] DIESTEL, R. *Graph Theory*. Springer-Verlag, 2000.

[7] DOAR, M. *Practical Development Environments*. O'Reilly Media, Incorporated, 2005.

[8] FOWLER, M., BECK, K., BRANT, J., OPDYKE, W., AND ROBERTS, D. *Refactoring: Improving the Design of Existing Code*. Pearson Education, 2012.

[9] GRAMA, A. *Introduction to parallel computing*. Pearson Education, 2003.

[10] HEINEMAN, G. T., POLLICE, G., AND SELKOW, S. *Algorithms in a Nutshell*. O'Reilly Media, Inc., 2009.

[11] HOFMANN, B., AND MARTIN, K. CMake Documentation. Retrieved from: http://www.cmake.org/cmake/project/about.html, Mar 2013.

[12] JOYNER, D., VAN NGUYEN, M., AND COHEN, N. Algorithmic graph theory. *Google Code* (2010).

[13] KERNIGHAN, B., AND PIKE, R. *The UNIX programming environment*. Prentice Hall, 1984.

[14] KERNIGHAN, B., AND PIKE, R. *The Practice of Programming*. Addison-Wesley, 1999.

[15] KERNIGHAN, B., AND RITCHIE, D. *The C programming language*. Prentice Hall, 1988.

[16] KNUTH, D. E. *Art of Computer Programming*, vol. I. Addison-Wesley, 1972.

[17] KWIATKOWSKI, P. "Implementation of the basic graph library in C". Master's thesis, Czestochowa University of Technology, Faculty of Mechanical Engineering and Computer Science, 2012.

[18] LONG, M. Unit Testing C code with the GoogleTest framework. Retrieved from: http://meekrosoft.wordpress.com/2009/11/09/unit-testing-c-code-with-the-google-test-framework/, Sep 2013.

[19] MARCH, S. T., AND SMITH, G. F. Design and natural science research on information technology. *Decision Support Systems 15*, 4 (1995), 251–266.

[20] MARTIN, R. C. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2008.

[21] OPENBSD. *OpenBSD Programmer's Manual. Malloc(3) manual page.*, Aug 2013.

[22] OTTO, M. OpenBSD's MALLOC_STATS and memory leak detection. Retrieved from: http://www.drijf.net/malloc/, Aug 2013.

[23] ROSEN, K. *Discrete Mathematics and Its Applications*. McGraw-Hill, 2012.

[24] SEDGWICK, R. *Algorithms in C*. Addison-Wesley, 1990.

[25] SIEK, J., LEE, L., AND LUMNSDAINE, A. The Boost Graph Library (BGL). Retrieved from: http://www.boost.org/doc/libs/1_51_0/libs/graph/doc/Graph.html, Nov 2012.

[26] WALLIS, W. *A Beginner's Guide to Graph Theory*. Birkhäuser Boston, 2007.

[27] WIRTH, N. *Algorithms and data structures*. Prentice-Hall London et al., 1986.

Index

assertions, 22
autotools, 7

Boost::Graph, 1, 2, 4, 24, 29–33, 35, 36
Breadth-First Search algorithm, 20, 21

CMake, 6, **7**, 7
CMake commands
    ccmake, 8
    cmake, 8
    cmake-gui, 8
CMake tutorial, 9
CMakeLists.txt, 8
CPack, 8
CTest, 8
    ctest, 24

DejaGNU, 2, 21
dependency tracking, 13, 15
Dijkstra's algorithm, 2, 6
Doxygen, 16

Edmonds-Karp algorithm, 2, 18

flow network, **17**
    augmenting paths, 17
    capacity, 17
    capacity constraints, 17
    flow conservation, 17
    residual capacity, 17
    skew symmetry, 17
    source, 17
    target, 17
Ford-Fulkerson algorithm, 2, **17**, 18

git, 3
GNU Public Licence, 3
Google C++ Mocking Framework, 23
Google C++ Testing Framework, **21**, 21
    a.k.a. googletest, 4, 21
    TEST() macro, 21
    TEST_T() macro, 21

igraph, 1, 24
in-source build type, 8

kconfig, 6
Kirchoff's law, 17

Linux Kernel List, 2, 20

make, 6–8, 12
Makefile, 7
maximum flow, 16

o-graph-ml, 3
out-of-source, 27
out-of-source build type, 8

pcmaker, 7
performance tests, 24

SCGL, v, 1–3, 5, 6, 14, 16, 21, 23, 24, 27–29, 31–33, 35–37, 39
SCons, 6
static edge attributes, 5
    capacity, 5
    cost, 5

unit tests, 21