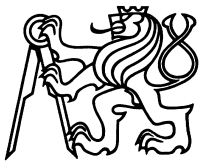
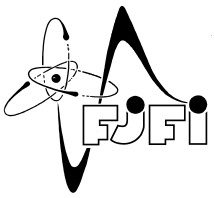


Czech Technical University in Prague
Faculty of Nuclear Sciences and Physical Engineering



Parallel graph algorithms for GPU

Paralelní grafové algoritmy pro GPU

Bachelor's Degree Project

Author: **Radek Cichra**

Supervisor: **doc. Ing. Tomáš Oberhuber, Ph.D.**

Language advisor: **Darren Copeland, MSc.**

Academic year: 2023/2024

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Cichra** Jméno: **Radek** Osobní číslo: **509217**

Fakulta/ústav: **Fakulta jaderná a fyzikálně inženýrská**

Zadávající katedra/ústav: **Katedra matematiky**

Studijní program: **Aplikovaná informatika**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Paralelní grafové algoritmy pro GPU

Název bakalářské práce anglicky:

Parallel graph algorithms for GPU

Pokyny pro vypracování:

1. Seznamte se se základy programování GPU pomocí CUDA a TNL.
2. Prostudujte současnou implementaci grafových algoritmů v knihovně TNL.
3. Implementujte grafové algoritmy pro výpočet souvislých a silně souvislých komponent, maximální nezávislé množiny a minimální kostry grafu.
4. Implementujte unit testy a benchmarky pro porovnání efektivity s jinými knihovnami jako například Boost, GraphBLAS a Gunrock.
5. Věnujte pozornost i refaktorování kódu a sepsání dokumentace s příklady použití implementovaných funkcí.

Seznam doporučené literatury:

[1] GPU programming guide - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
[2] TNL Users' guide - https://tnl-project.gitlab.io/tnl/UsersGuide.html
[3] Kepner J., Gilbert J., Graph Algorithms in the Language of Linear Algebra, Society for Industrial and Applied Mathematics, 2011.
[4] Zhang Y., Azad A., Buluç A., Parallel algorithms for finding connected components using linear algebra, Journal of Parallel and Distributed Computing, vol. 144, pp. 14-27, 2020.
[5] Baer T., Kanakagiri R., Solomonik E., Parallel Minimum Spanning Forest Computation using Sparse Matrix Kernels, Proceedings of the 2022 SIAM Conference on Parallel Processing for Scientific Computing (PP), pp. 72-83, 2022.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

doc. Ing. Tomáš Oberhuber, Ph.D. katedra matematiky FJFI

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **31.10.2023** Termín odevzdání bakalářské práce: **05.08.2024**

Platnost zadání bakalářské práce: **30.09.2025**

doc. Ing. Tomáš Oberhuber, Ph.D.	prof. Ing. Zuzana Masáková, Ph.D.	doc. Ing. Václav Čuba, Ph.D.
podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

28/11/2023

Datum převzetí zadání

Podpis studenta

Acknowledgment:
I would like to thank doc. Ing. Tomáš Oberhuber, Ph.D. for his expert guidance and express my gratitude to Darren Copeland, MSc. for his language assistance.

Author's declaration:
I declare that this Bachelor's Degree Project is entirely my own work and I have listed all the used sources in the bibliography.

Prague, August 5, 2024 Radek Cichra

Název práce:

Paralelní grafové algoritmy pro GPU

Autor: Radek Cichra

Studijní program: Aplikovaná informatika

Druh práce: Bakalářská práce

Vedoucí práce: doc. Ing. Tomáš Oberhuber, Ph.D., Katedra softwarového inženýrství, Fakulta jaderná a fyzikálně inženýrská, České vysoké učení technické v Praze

Abstrakt: Tato bakalářská práce je zaměřená na několik běžných grafových problémů a na jejich řešení s pomocí paralelních grafových algoritmů. Jedná se o problémy nalezení maximální nezávislé množiny, minimální kostry grafu, souvislých a silně souvislých komponent. Tato práce poskytuje teoretický kontext potřebný pro popsání zmíněných problémů a navrhuje jejich řešení za pomoci běžných sekvenčních algoritmů. Hlavní snaha práce však směřuje k algoritmům paralelním a jejich implementaci využívající CUDA, C++ a knihovnu TNL. Tyto implementace jsou schopny běhu jak na GPU paralelně, tak na CPU sekvenčně (či paralelně s pomocí OpenMP). Práce obsahuje detailní popis těchto algoritmů a jejich implementace. Také informuje o procesu testování správné funkčnosti, měření výkonu na různých zařízeních (GPU, CPU) a uprávách vedoucích k lepšímu výkonu. Výstupem práce jsou plně funkční a otestované implementace pro 3 ze 4 problémů. Pro problém minimální kostry grafu je výstup rozpracovaná částečně fungující implementace.

Klíčová slova: Benchmarkování, CUDA C++, GPU Programování, Knihovna TNL, Maximální Nezávislá Množina, Minimální Kostra Grafu, OpenMP, Paralelní Algoritmy, Silně Souvislá Komponenta, Souvislá Komponenta, Teorie Grafů, Unit Testování, Výpočetní Cluster

Title:

Parallel graph algorithms for GPU

Author: Radek Cichra

Abstract: This bachelor thesis targets several common graph problems and their solutions using parallel graph algorithms. These are the problems of finding the maximal independent set, minimal spanning tree, connected and strongly connected components. This paper provides the theoretical context needed to describe the aforementioned problems and proposes their solutions using common sequential algorithms. However, the main effort of the thesis is directed towards parallel algorithms and their implementation using CUDA, C++, and the TNL library. These implementations are capable of running both on the GPU in parallel and on the CPU sequentially (or in parallel using OpenMP). This thesis contains a detailed description of these algorithms and their respective implementations. It also reports on the process of correctness testing, performance measurements on different devices (GPU, CPU), and further improving performance. The output of this work is fully functional and tested implementations for 3 of the 4 problems. For the minimal spanning tree problem, a semi-functional work-in-progress implementation is offered.

Key words: Benchmarking, Connected Component, CUDA C++, GPU Programming, Graph Theory, High Performance Computing, Maximal Independent Set, Minimal Spanning Tree, OpenMP, Parallel Algorithms, Strongly Connected Component, TNL Library, Unit Testing

Contents

Introduction

The field of graph theory is a crucial part of modern computer science and many other seemingly unrelated fields. This is due to its immense versatility and ability to model various problems. Graphs are without a doubt an extremely useful tool. It is then no surprise that the field is well studied and actively developing. An obvious way to move forward is to employ a parallel approach and focus on parallel solutions to graph problems. This has been, of course, the direction in which the field is moving for some time now.

The goal of this thesis is to select, describe, test, implement, and benchmark parallel algorithms for solving common graph problems - finding maximal independent set, (strongly) connected components, and minimal spanning tree (forest). Final solutions are to be implemented as a part of TNL library and utilize parallel computing on a GPU to be more efficient.

The first chapter lays out graph theory, TNL library, and GPU programming background relevant to this work. Following chapters (2 - 5) each focus on a given graph problem and its solution. Chapters 6 and 7 comment on unit testing, the benchmarking process, and results, followed by conclusion and discussion regarding the thesis as a whole.

Chapter 1

Preliminaries

1.1 Graphs

This section briefly lays out the terminology and explains terms later used when formulating graph problems and their solutions.

(Directed) Graph

A graph is an ordered set $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ where $\mathbf{V}$ is a set of vertices. An element of $\mathbf{E}$, called an edge, is an *ordered* unique pair of vertices $(u, v) \mid u, v \in \mathbf{V}$. This edge denotes a directed connection from u to v.

Undirected Graph

An undirected graph is defined as a graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ where $\mathbf{E}$ contains *unordered* unique pairs of vertices $\{u, v\} \mid u, v \in \mathbf{V}$. This means that an edge $\{u, v\} \in \mathbf{E}$ denotes a bi-directional connection between u and v. If $\{u, v\} \in \mathbf{E}$, then there is an edge from u to v and v to u.

Weighted Undirected Graph

A weighted undirected graph is an ordered set $\mathbf{G} = (\mathbf{V}, \mathbf{E}, w : \mathbf{E} \to \mathbb{N}^+)$ where $\mathbf{V}$ is a set of vertices, $\mathbf{E}$ is a set of undirected edges, and w is a weight function. w is a projection of edges onto positive integers, assigning each edge $\{u, v\} \in \mathbf{E}$ weight $w(\{u, v\}) \in \mathbb{N}^+$.

Graph Weight Sum

Suppose there is a weighted undirected graph $\mathbf{G} = (\mathbf{V}, \mathbf{E}, w : \mathbf{E} \to \mathbb{N}^+)$. Let us define $e = \{u, v\}$ and $\sum_{e \in \mathbf{E}} w(e) = \mathbf{W}$ as the sum of weights of all edges on $\mathbf{G}$.

Vertex Adjacency

Suppose there is an undirected graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$. Vertices $u, v \in \mathbf{V}$ are *adjacent* (neighbours) if and only if they are connected by a single direct edge $\{u, v\} \in \mathbf{E}$. We show this using the notation $u \sim v$.

Vertex Reachability

Suppose there is a directed or undirected graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$. Vertices $u, v \in \mathbf{V}$ are mutually *reachable* if and only if there exists a sequence of edges $((u, a), (a, b), \ldots, (x, v)) \in \mathbf{E}$. We consider every vertex to be reachable by itself.

Vertex Degree

Suppose there is a directed or undirected graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$. For every vertex $n \in \mathbf{V}$, we define degree $d(n)$ as the **number of vertices adjacent to it** (ie. reachable over a single direct edge).

Tree

Suppose there is an undirected graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$. $\mathbf{G}$ is a tree if and only if for any two vertices u, v, there exists **exactly one** path (sequence of edges) that connects them.

Spanning Tree

Suppose there is an undirected graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$. A subgraph $\mathbf{T} = (\mathbf{V}', \mathbf{E}')$ is a spanning tree if and only if it is a tree and $\mathbf{V} = \mathbf{V}'$.

Adjacency Matrix

Let us represent graph $\mathbf{G}$ with n vertices by a matrix $\mathbf{A_G}$ of size $n \times n$ where $\mathbf{A_{uv}} \neq 0$ if and only if $(u, v) \in \mathbf{E}$. For a weighted graph $\mathbf{H} = (\mathbf{V}, \mathbf{E}, \ w : \mathbf{E} \to \mathbb{N}^+)$, the value $\mathbf{A_{uv}} = i$ denotes the weight of $(u, v) \in \mathbf{E}$.

1.2 GPU

GPU (Graphics Processing Unit) is a hardware component initially developed for accelerated image rendering and computer graphics. While CPUs have been optimized extensively for mostly sequential execution with highly sophisticated tools such as branch predicting, GPUs excel at parallel workloads. This is because GPUs allocate much more of their resources to the actual data processing, at the cost of finer control over each individual core it has (see figure 1.1). This makes GPUs ideal for executing thousands of threads at once in parallel [Nvi24]. They can be heavily utilized when solving problems that are *embarrassingly parallel* (well suited for parallel execution).

Note that a GPU, being its own hardware component, usually has its own onboard memory and memory space separated from that of a CPU (DRAM and cache memory in figure 1.1). This is important to consider, as to reap the benefits of executing code in parallel on the GPU, you want to mostly work within this space. Otherwise, you will face the slowdown caused by frequent communication between CPU and GPU, which is much slower than just local communication between parts of GPU.

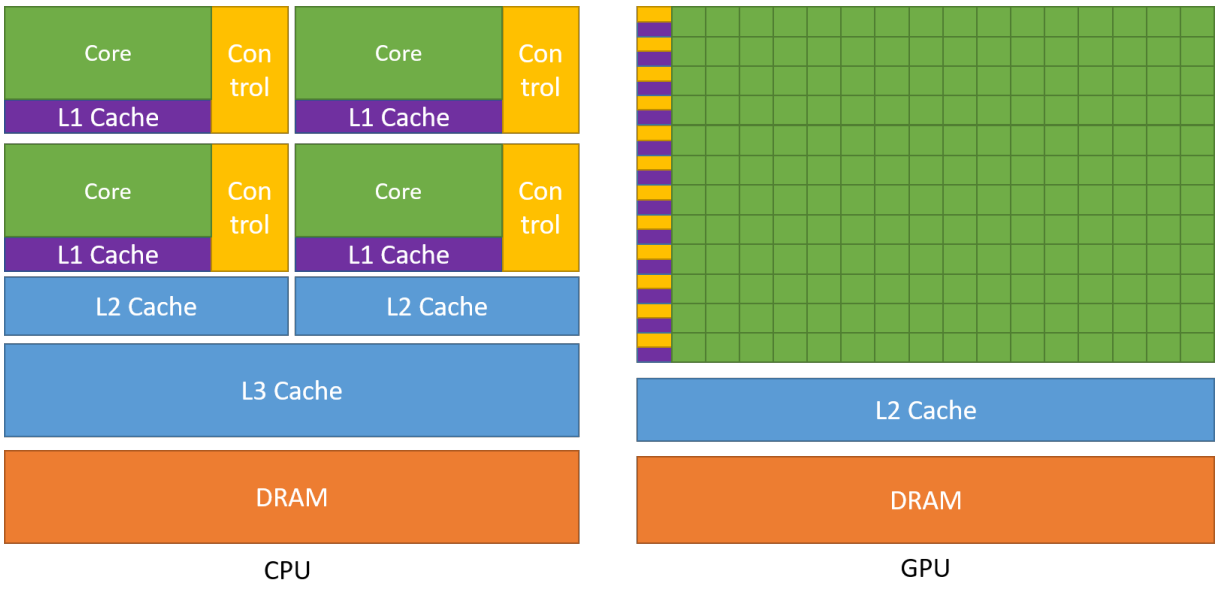


Figure 1.1: Comparison of CPU and GPU architecture. Note the much higher number of cores a single control unit oversees in GPUs [Nvi24]

1.3 CUDA

CUDA is a general-purpose parallel computing platform and programming model made by Nvidia for **Nvidia GPUs** [Nvi24]. Its goal is to allow, simplify, and improve the development of applications leveraging parallelism. It supports several high-level programming languages, including C++. CUDA abstracts technical details, such as hardware specifications, to create a programming model with a low learning curve. It also allows for scalability and enables a compiled CUDA program to **execute on any number of streaming multiprocessors**.

When working with CUDA, the user will write a kernel to be executed on the GPU. This is simply a function that, when called, is executed **N** times in parallel by **N** CUDA threads. These threads have unique IDs, which can be used for directly assigning tasks to them. Having this sort of access to them allows us to essentially *map* our problems onto our GPU's computing units. Threads can be grouped into blocks with shared memory. These blocks can in turn also be grouped into block clusters with shared memory, which lay on a cluster grid with the global memory (see figure 1.2).

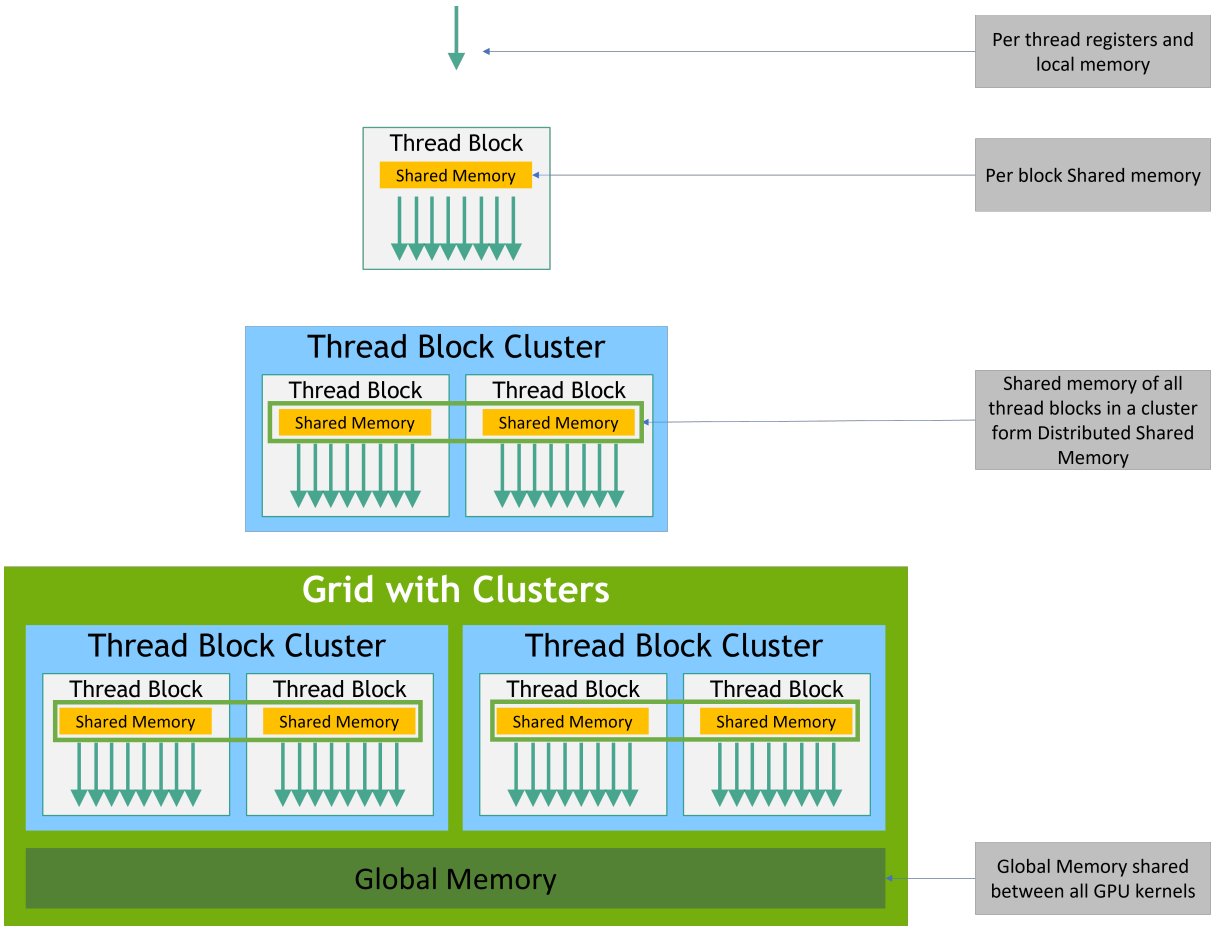


Figure 1.2: CUDA Memory Hierarchy [Nvi24]

1.4 TNL

TNL is an open-source library with the goal of creating a user-friendly and flexible toolkit for the development of efficient numerical solvers and HPC algorithms. It is predominantly written in C++ and leverages templates, lambda functions, abstraction, and other modern paradigms. It also provides native support for modern hardware architectures and allows the user to manage and write code for different memory spaces through a single unified interface. In turn, this allows for the creation of code capable of running on multiple hardware architectures and memory spaces, all of which remains mostly hidden from the user behind a layer of abstraction, making the work easier and much less tedious. In an ideal scenario, this allows the user to write hardware-agnostic code. TNL supports parallelism on both GPU (using CUDA, HIP, or ROCm) and CPU (using OpenMP) [OKF21] [KOFv22].

1.4.1 TNL data structures

TNL contains a vast number of structures for storing data. Some structures relevant in the context of this work are:

- `TNL::Containers::Vector`
- `TNL::Matrices::SparseMatrix`

- `TNL::Graphs:Graph`

These are templated classes defined within TNL and are used in the algorithm implementations described in this work.

`Vector` has 3 template parameters:

- `Real` being the type of data stored in `Vector`
- `Device` being the device where the `Vector` is allocated (`Sequential` or `Host` for CPU, `CUDA` for GPU)
- `Index` being the type used for indexing the elements of `Vector`

`SparseMatrix` is one of two main matrix representations in TNL (other one being `DenseMatrix`). As the name suggests, it is aimed at storing matrices with a small number of non-zero elements relative to the matrix size. In TNL, various formats for the storage of `SparseMatrix` non-zero segments exist. When working with sparse matrices, using `SparseMatrix` is more efficient in terms of storage.

Since our work revolves around graphs, and their matrix representation is *usually* sparse, we decided to focus on this data structure in our implementations. Furthermore, code written to work with `SparseMatrix` representation should also handle `DenseMatrix`.

`Graph` has 2 template parameters:

- `Matrix` being the type of `Matrix` used for the representation of the graph.
- `GraphType` being the type of graph (`Undirected` or `Directed`)

1.4.2 TNL Views

Another important part of TNL worth talking about is the way it handles passing data between devices. Since we are free to allocate objects to different memory spaces (see section 1.2), the need for handling cross-device references arises. Deep copy of said object could simply be created on the device we want to access it from but this is very slow and won't update the original on the other device. TNL overcomes this issue via `View` types defined for its containers. `View` is a kind of lightweight reference object which makes only a shallow copy of itself in copy constructor. It can be captured *by value* in lambda functions, making it the primary (and for us only) means of modifying data of a given object on a different device.

For most intents and purposes, `View` serves the role of a reference to an object across devices and can be used to change the object. Copying `View` between devices is much cheaper than doing the same with the whole object. This allows for a very efficient way of working across memory spaces. User must only ensure that the `View` is still valid and referencing said object. Causing re-allocation, for example when resizing the object, invalidates its `View` [Tem24].

1.4.3 Lambda functions

The process of writing code for the GPU and for the CPU is vastly different in general. For GPUs, we must often write lengthy kernels with a lot of necessary code to configure them. This adds a lot of

lines to it every time we need to sum up some data, for example. Debugging kernels is also a chore.

As was already stated, TNL aims to abstract as many differences between hardware architectures and memory spaces from the user as possible. One way they do this is by offering skeletons or patterns of frequently used routines or functions. These are then combined with user defined lambda functions to perform even fairly contextual tasks with ease. Writing a lambda function is much less tedious than configuring a kernel. Furthermore, code implemented using this approach can be executed on both GPU and CPU. This also allows for easier debugging, as we can fall back onto the CPU execution for it. Then, modifying the code to work on both devices usually requires little to no changes. Note that inside of lambda functions, we must use `View` of an object to interact with it.

The main routine skeleton (in `TNL::Algorithms` namespace) relevant to this work is `parallelFor`. It is a parallel for-loop function for one dimensional range with integer indexing. A general way of using it would be

```
parallelFor< Device >( begin, end, f );
```

where `Device` specifies whether the code is executed on the GPU or CPU, `begin` and `end` denote the range on which `parallelFor` is performed, and `f` is the user-defined lambda function. Based on `Device`, this routine is then called from either a CUDA kernel, processed by OpenMP threads, or done sequentially. The following code snippet tries to show a common usage of `parallelFor` - performing an operation on vectors in parallel.

```
/*
...
TNL vectors 'v1', 'v2', and 'result' defined with same size
'Device' defined
...
*/
auto size = v1.getSize();
//fetching views
auto v1_view = v1.getView();
auto v2_view = v2.getView();
auto result_view = result.getView();

auto f = [=] __cuda_callable__ (int i) mutable {
    result_view[i] = v1_view[i] + v2_view[i];
};
TNL::Algorithms::parallelFor<Device>(0,size,f);
```

Here, `__cuda_callable__` is an *umbrella term* defined by TNL. When creating a lambda function, we need to specify the `Device` it should be executed on. Since we aim for all devices, this term covers all 3 - `Sequential`, `Host`, and `CUDA`. Based on what `Device` we provide in the `parallelFor` call, an appropriate term is slotted in.

`reduce` is another commonly utilized routine. It is a way to reduce some set of elements into just a single one using user-specified operation on per-element basis. It is usually called like

```
reduce< Device >( begin, end, fetch, reduction, id );
```

where `Device`, `begin`, and `end` serve the same purpose as for `parallelFor`. User-defined lambda functions `fetch` and `reduction` specify the way we want to capture values and the reduction operation used on them, respectively. When using custom reduction functions, we also need to specify the identity

element `id`. It simply denotes the identity for given reduction operation, such as 1 for multiplication or 0 for addition. In the snippet below you can see how reduction would be used to sum vector elements.

```
/*
...
'v1' and 'v1_view' defined
'Device' defined
...
*/
auto size = v1.getSize();
auto fetch = [=] __cuda_callable__ (int i) -> double {
    return v1_view[i];
};

auto reduce = [] __cuda_callable__ (const double& a, const double& b) {
    return (a + b);
}
double sum = TNL::Algorithms::reduce<Device>(0,size,fetch,reduce,0);
```

TNL also contains some baked-in reductions, such as `TNL::Plus{}`, `TNL::Min{}`, `TNL::Max{}`, and others with identity elements already specified, so you can simply call them like

```
reduce< Device >( begin, end, fetch, predefined_reduction );
```

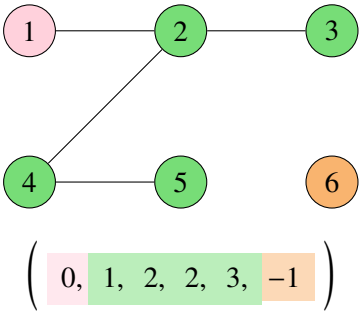
1.4.4 Breadth first search

Later in our implementation, we lean on some graph-specific functions and routines already present in the TNL library. One such function, utilized in section 3.5, is **Breadth-first-search** [HT73], also referred to as **BFS**. In the context of our work, it is used to traverse and explore graphs from a given starting vertex. Note that **Depth-first-search** could be used in its place, but due to it being inherently sequential [Rei85], we are much better off with **BFS**. In TNL, it is in `TNL::Graphs` namespace and is called like

```
breadthFirstSearch( graph, start, distances );
```

where `graph` is self explanatory, `start` is index of the vertex we want to start searching from. Results are stored in `distances` vector in the form of relative distance of each vertex from the `start` vertex. For unweighted graphs, each edge gets a weight of 1. The `start` vertex itself gets a distance of 0 in the `distances` vector. Unreachable vertices get -1. It implicitly follows that the size of `distances` vector must be equal to the number of vertices of `graph` (see figure 1.3).

Figure 1.3: Running **BFS** on the following graph, starting from vertex 1, results in the `distances` vector shown below



In our code, we use `breadthFirstSearchTransposed`, which is a slightly modified version of the `breadthFirstSearch` function. It takes the internal adjacency matrix representation of the graph as a parameter.

Chapter 2

Connected Components

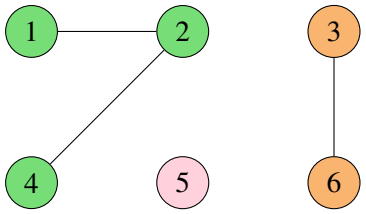
2.1 Theory

Suppose an undirected graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$. Connected Component **CC** is defined as a subset of **V** in which all vertices are mutually reachable. This can be expressed as

$$\mathbf{CC} \subseteq \mathbf{V} \text{ where } \forall u, v \in \mathbf{CC} \mid (\{u, x_1\}, \{x_1, x_2\} \ldots \{x_n, v\}) \in \mathbf{E}$$

We also require that $\forall u \in \mathbf{CC}$, there are no reachable vertices that are outside of **CC**. Using this definition, finding connected components is equivalent to uniquely partitioning the graph into its maximal connected subgraphs (see figure 2.1).

Figure 2.1: Color coding valid **connected components** of a graph.



2.2 Problem definition

Identify all connected components of an undirected unweighted graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$.

2.3 Sequential algorithm

Identifying all connected components of a graph sequentially can be done very easily using either a **depth-first-search** or a **breadth-first-search** [HT73]. Such an algorithm can look as follows:

Algorithm 1 Sequential CC
Input: undirected graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$
Output: set of connected components **CC**

Initialize $\mathbf{CC} = \emptyset$
while $V \neq \emptyset$ **do**
  select a vertex $n \in V$
  explore all vertices reachable from n using depth/breadth-first-search
  $\mathbf{cc_n} = \mathbf{BFS}(\mathbf{G}, \mathbf{n})$ (**BFS** on **G** starting from n)
  add $\mathbf{cc_n}$ to **CC**
  remove $\mathbf{cc_n}$ from V
end while
return **CC**

Inner workings of the algorithm are very much apparent and intuitive. Both depth and breadth first search end up fully exploring vertices reachable from a given vertex n, therefore each iteration finds a connected component to which n belongs. By then removing our newfound component from available vertices, we ensure that in next iteration, a vertex from another component will be selected. Once we run out of vertices, we return vector of connected components **CC**.

2.4 Parallel algorithm

For our implementation, we propose a version of **FastSV Algorithm** [ZAH20], [ZAB20] which is an efficient simplification of the much older **SV Algorithm** [SV82].

Algorithm 2 FastSV
Input: undirected graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$
Output: component vector **p**

Initialize vectors **p**, **gp** to track parents and grandparents of each vertex
Assume $\forall n \in V$ to be rooted tree $\rightarrow \mathbf{p_n} = \mathbf{gp_n} = \mathbf{n}$
repeat
  for all $\{u, v\} \in E$ (in parallel) **do**
    if $\mathbf{gp_u} > \mathbf{gp_v}$ **then**
      set $\mathbf{p_{p_u}} = \mathbf{gp_v}$ (stochastic hooking)
      set $\mathbf{p_u} = \mathbf{gp_v}$ (agressive hooking)
    end if
  end for
  for all $n \in V$ (in parallel) **do**
    if $\mathbf{p_n} > \mathbf{gp_n}$ **then**
      set $\mathbf{p_n} = \mathbf{gp_n}$ (shortcutting)
    end if
  end for
  for all $n \in V$ (in parallel) **do**
    set $\mathbf{gp_n} = \mathbf{p_{p_n}}$ (updating **gp**)
  end for
until vector **gp** stops changing
return **p**

The core idea behind FastSV is as follows - let us represent each connected component of a graph using a rooted tree, where the root of each component is the lowest index vertex in it. We represent this by a parent vector $\mathbf{p}$, where $\mathbf{p_n}$ denotes a parent of vertex n, and root vertices are their own parents (see figure 2.2). We also define a grandparent vector $\mathbf{gp}$, where $\mathbf{gp_n} = \mathbf{p_{p_n}}$.

Initially, knowing nothing about the graph, we assume every vertex to be its own connected component, making them all root vertices (line 2). Then, we repeat 3 steps until convergence - *hooking*, *shortcutting*, and *updating*.

In the hooking step, we join (hook) together vertices that are part of the same connected component. By hooking u to v, we mean changing the parent ($\mathbf{p_u}$) and grandparent ($\mathbf{p_{p_u}}$) of u to a grandparent ($\mathbf{gp_v}$) of v (lines 6 and 7). Let us now elaborate on why we do this. First of all, note that hooking is performed in parallel over all edges (line 4) - this means that we only ever try to hook vertices that are actually reachable, therefore part of the same component. Now for the hooking condition (line 5) - our goal is to hook all vertices of a component to the lowest index vertex in it. By hooking u to v only when $\mathbf{gp_u} > \mathbf{gp_v}$, we make sure to always hook higher index vertex to a lower index one. In combination with shortcutting, this eventually leads to every vertex of a component hooking to the lowest index one.

In shortcutting step (lines 10-14), we go over all vertices in parallel and try to reduce tree height of our current connected components represented by rooted trees in the $\mathbf{p}$ vector. If a vertex n has a lower index grandparent than parent, we can *move it up a level* by making $\mathbf{p_n} = \mathbf{gp_n}$. This reduces the height of trees that represent our connected components, which is crucial, as it allows for finding new smaller index vertices to hook in the next iteration.

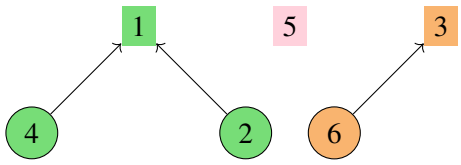
In update step (lines 15-17), we simply update $\mathbf{gp}$ to reflect the hooking and shortcutting of this iteration.

These steps are repeated, until $\mathbf{gp}$ converges, at which point vector $\mathbf{p}$ represents a collection of stars corresponding to connected components (see figure 2.2). This is because of the shortcutting, which flattens the trees so that vertices can find smaller and smaller indexed parents, until they all share the smallest indexed vertex in a component.

To reiterate, we start with each vertex being its own component (line 2). We then find and hook each vertex to a smallest index one we can reach and see (lines 4-9). This creates newly joined components represented by trees in $\mathbf{p}$, which we try to flatten whenever we can (lines 10-14). We then update the $\mathbf{gp}$ vector to reflect our changes (lines 15-17), and repeat the process from hooking onward. Once the $\mathbf{gp}$ converges, we can return $\mathbf{p}$ as our result, representing all connected components of the graph.

Note that the decision to converge with smallest indexed vertices as roots of the components is arbitrary. The algorithm can be adapted to converge with the largest indexed vertices instead. We could also use the $\mathbf{p}$ vector for the convergence condition, but it can be shown that checking $\mathbf{gp}$ is also valid and can actually prevent one last redundant iteration of the algorithm loop [ZAB20].

Figure 2.2: *Pointer graph* representation of vector $\mathbf{p} = [\mathbf{1}, \mathbf{1}, \mathbf{3}, \mathbf{1}, \mathbf{5}, \mathbf{3}]$ of the graph shown in figure 2.1. Every component is a star with its lowest index vertex being the root (its own parent).



2.5 TNL implementation

We implement FastSV as a function `TNL::Graphs::connectedComponents` (also referred to as `CC_opt2` in subsections 7.5.1 and 7.6.1) defined with the following parameters:

```
template< typename GraphType, typename OutType >
void connectedComponents( const GraphType& g, OutType& ccV){/*code*/}
```

where `g` is an undirected graph and `ccV` is the vector to which we will save our results. Note that size of `ccV` must be equal to the number of vertices of `g`. Our implementation has two main parts - *declarative* and *main iterative*.

```
/*declarative part*/
//fetching type info
using DeviceType = typename GraphType::DeviceType;
using IndexType = typename GraphType::IndexType;
using IndexVectorType = TNL::Containers::Vector< IndexType, DeviceType, IndexType >;

const IndexType size = g.getNodeCount();

//fetching views of the input variables
auto ccVview = ccV.getView();
auto& adjMatrix = g.getAdjacencyMatrix();
const auto adjMatrixView = adjMatrix.getConstView();

OutType p( size );      //parent vector
OutType gp( size );     //grandparent vector
OutType gpOld( size );  //cpy of gp for convergence check
//views
auto pView = p.getView();
auto gpView = gp.getView();
auto gpOldView = gpOld.getView();

//# of edges - 2 counts per edge (each direction = 1)
auto totalEdges = adjMatrix.getSegments().getStorageSize();

IndexVectorType rowCapacities( size );      //# of neighbours of each vertex
adjMatrix.getRowCapacities( rowCapacities );//fetch the capacities

//vector holding indices partitioning the list of all edges
//so that each vertex has its own partition
IndexVectorType indices( size, -1 );
indices = rowCapacities;

//compute starting indices
```

```
34 TNL::Algorithms::inplaceExclusiveScan( indices );
35 auto indicesView = indices.getView();
36
37 //vectors to store edges
38 IndexVectorType fromVector( totalEdges, -1 );
39 auto fromVectorView = fromVector.getView();
40 IndexVectorType toVector( totalEdges, -1 );
41 auto toVectorView = toVector.getView();
42
43 //initializion + fetching edges
44 auto init = [ = ] __cuda_callable__( IndexType rowIdx ) mutable
45 {
46     IndexType starting_index = indicesView[ rowIdx ];
47
48     const auto row = adjMatrixView.getRow( rowIdx );
49     for( IndexType i = 0; i < row.getSize(); i++ ) {
50         fromVectorView[ starting_index + i ] = rowIdx;
51         toVectorView[ starting_index + i ] = row.getColumnIndex( i );
52     }
53
54     pView[ rowIdx ] = rowIdx;
55     gpView[ rowIdx ] = rowIdx;
56     gpOldView[ rowIdx ] = rowIdx;
57 };
58 TNL::Algorithms::parallelFor< DeviceType >( 0, size, init );
```

In the declarative part, we first retrieve type information (`DeviceType` and `IndexType`) from templates. We then note some important characteristics of `g` for later use, namely:

- `size` - number of vertices
- `adjMatrix` - adjacency matrix representation
- `rowCapacities` - number of outgoing edges from each vertex
- `totalEdges` total number of edges (counting each direction as a single edge)

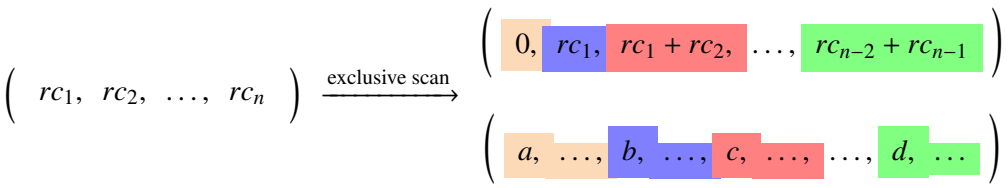
We use `size` to declare vectors for tracking the (grand)parents of vertices during the algorithm - `p`, `gp`, and `gpOld`.

We use `rowCapacities` and `totalEdges` to create an ordering of all edges, which we store in two vectors - `fromVector` and `toVector` - of size `totalEdges`, where

$$\texttt{fromVector[i]} = u \wedge \texttt{toVector[i]} = v \iff (u, v) \in \mathbf{E} \text{ and is } \mathbf{i^{th}} \text{ in our ordering}$$

To create such an ordering, we need to map all edges of `g` onto our vectors. If we perform exclusive scan on `rowCapacities` (storing results in `indices` to not modify `g`), we receive a way to partition our edge vectors by vertices (see figure 2.3).

Figure 2.3: Exclusive scan on `rowCapacities` vector (left) leads to a partition vector `indices` (top right). This vector contains the beginning index for each vertex edge list segment in the edge vectors (bottom right). First edge segment begins at index a and has a size of rc_1. Second segment follows and ends at index $rc_1 + rc_2$ (ie. has size of rc_2), and so on...



Knowing this, we can fetch all the edges in the `init` lambda function. While this forces a sequential loop to find all edges, it has to be done only once here, whereas the alternative would be forcing a sequential loop later during each iterative step. Furthermore, we need to run `init` lambda at least once anyway to initialize (grand)parent vectors.

```
/*main iterative part*/
do {
    gpOldView = gpView;

    //stochastic and agressive hooking
    auto hooking = [ = ] __cuda_callable__( IndexType Idx ) mutable
    {
        if( gpView[ fromVectorView[ Idx ] ] > gpView[ toVectorView[ Idx ] ] ) {
            pView[ pView[ fromVectorView[ Idx ] ] ] = gpView[ toVectorView[ Idx ] ];
            pView[ fromVectorView[ Idx ] ] = gpView[ toVectorView[ Idx ] ];
         }
    };
    TNL::Algorithms::parallelFor< DeviceType >( 0, totalEdges, hooking );

    //shortcutting
    auto shortcutting = [ = ] __cuda_callable__( IndexType rowIdx ) mutable
    {
        if( pView[ rowIdx ] > gpView[ rowIdx ] ) {
            pView[ rowIdx ] = gpView[ rowIdx ];
        }
    };
    TNL::Algorithms::parallelFor< DeviceType >( 0, size, shortcutting );

    //grandparent update
    auto gpUpdate = [ = ] __cuda_callable__( IndexType rowIdx ) mutable
    {
        gpView[ rowIdx ] = pView[ pView[ rowIdx ] ];
    };
    TNL::Algorithms::parallelFor< DeviceType >( 0, size, gpUpdate );

} while( gpView != gpOldView );
ccVview = pView;
```

In the main iterative part of the implementation, we perform hooking, shortcutting, and updating of `gp`, each in their respective lambda function. We repeat these steps until `gp` converges. Note the parallelism over all edges for the `hooking` lambda - without establishing `fromVector` and `toVector`, there would

be an already mentioned forced sequential loop, as we would have to iterate in parallel over vertices instead.

Once the grandparent vector `gp` stops changing, we terminate the iterative step, copying the parent vector `p` into the result vector `ccV`.

Chapter 3

Strongly Connected Components

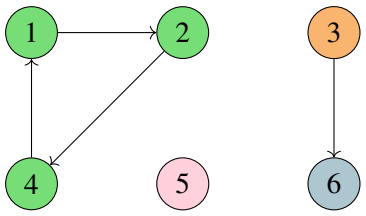
3.1 Theory

Suppose a directed graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$. Strongly connected component **SCC** is a subset of **V** where all vertices are mutually reachable. In other words

$$\mathbf{SCC} \subseteq \mathbf{V} \text{ where } \forall u, v \in \mathbf{SCC} \mid ((u, x_1), (x_1, x_2) \dots (x_n, v), (v, y_1), (y_1, y_2) \dots (y_n, u)) \in \mathbf{E}$$

We assume every vertex to be reachable by itself, as well as full exploration of any paths leading to and from any given vertex to ensure **SCCs** being maximal. Note that for any two vertices to be mutually reachable, they must both be in a cyclic subgraph containing paths that connect them. Similar to connected components on undirected graphs, strongly connected components uniquely partition a given graph into parts (see 3.1).

Figure 3.1: Color coding valid **strongly connected components** of a graph.



3.2 Problem definition

Identify all strongly connected components on a directed unweighted graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$.

3.3 Sequential algorithm

Sequentially finding strongly connected components of a directed graph can be done with rather well-known **Tarjan's SCC algorithm** [Tar72], or with simpler (but slower) **Kosaraju-Sharir's algorithm** [Sha81]. For convenience (and its similarity to our parallel algorithm), let us focus on the latter one.

First we have to introduce the notion of *transposed graph*. Assuming a directed graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$,

we define its transposed graph $\mathbf{G^T} = (\mathbf{V}, \mathbf{E^T})$ as a graph with the same vertices and *inverted* edges (ie. inverted edge direction). Bi-directional edges stay the same. Knowing this, we can describe the algorithm as follows:

Algorithm 3 Sequential SCC
Input: directed graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$
Output: set of strongly connected components **SCC**

```
1: initialize SCC = ∅
2: construct G^T = (V, E') (transpose graph)
3: while V ≠ ∅ do
4:     select a pivot vertex n ∈ V
5:     explore vertices reachable from n on G using breadth-first-search
6:     desc = BFS(G, n)
7:     explore vertices reachable from n on G^T using breadth-first-search
8:     pred = BFS(G^T, n)
9:     vertices found in both are part of a strongly connected component
10:    scc_n = desc ∩ pred
11:    remove scc_n from G and G^T
12: end while
13: return SCC
```

This algorithm hinges on the fact, that overlaying vertices reachable from n (**desc**) with vertices, from which n is reachable (**pred**), results in a strongly connected component. This should intuitively make sense, as in order for the vertices to be mutually reachable, we must be able to both reach them from n, as well as reach n from them. In essence, this is equal to stating that all the vertices must be part of a cyclic subgraph, so that any two vertices on it are reachable (see figure 3.3 for visual representation). This is exactly the definition of a strongly connected component.

Because the graph is partitioned by strongly connected components, we can base the terminating condition on their removal.

3.4 Parallel algorithm

For finding strongly connected components, we propose a modified version of the **Divide-and-conquer Algorithm for Identifying SCCs** proposed by Coppersmith, Fleisher, Hendrickson, and Pinar [CFHP06]. The main modifications we've made was removing recursion and implementing *coloring*.

By coloring, we simply mean giving all vertices, that make up one strongly connected component, one **unique positive integer identifier**, also known as **color**. We do this so that we can more easily store and express all the strongly connected components of a graph - since each component is made of vertices with one unique color, we just need to provide a vector of all vertices with their corresponding colors to show all strongly connected components (see figure 3.2).

Figure 3.2: Vector showing colored vertices of a graph in figure 3.1. Index of the vector element denotes the vertex, and the element value its color.

$$\mathbf{scc} = \begin{pmatrix} 1, & 1, & 2, & 1, & 3, & 4 \end{pmatrix}$$

Algorithm 4 CFHP (*modified for TNL*)
Input: directed graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$
Output: colored component vector **scc**

```
1:  initialize vector scc and color c to track colored vertices
2:  repeat
3:     select an uncolored vertex n ∈ V
4:     find predecessors and descendants of n → pred, desc
5:     for all u ∈ V (in parallel) do
6:        if u ∈ pred ∧ u ∈ desc then
7:           set scc_u = c
8:        end if
9:     end for
10:    increase c
11: until ∀v ∈ scc are colored
12: return scc
```

CFHP algorithm starts by initializing an *uncolored* vector **scc** representing the vertices of graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ and a color c for coloring. We assume color of 0 to represent *uncolored* state and initialize color c to 1.

In its main iterative step, we select an uncolored vertex $n \in$ **scc** and explore (via breadth-first-search) all the vertices reachable from n (descendants) as well as all vertices from which n is reachable (predecessors). This is done by exploring both the original graph, as well as the transpose graph. We note these vertices (including n) in their respective boolean vectors - **desc** and **pred**, where

$$\mathbf{pred_i} = \mathbf{true} \iff n \text{ is reachable from } i$$

$$\mathbf{desc_i} = \mathbf{true} \iff i \text{ is reachable from } n$$

Then we compare these vectors to find vertices which appeared in both and color them with c. By doing this, we have found and colored a strongly connected component. Last thing we do is increase c for the next iteration (next component).

Algorithm terminates, once **scc** is fully colored.

3.5 TNL implementation

We implement CFHP algorithm as a function `TNL::Graphs::scc` defined with the following parameters:

```
template< typename GraphType, typename OutType >
void scc( const GraphType& g, OutType& coloredSCCs ){/*code*/}
```

where `g` is our graph and `coloredSCCs` is the result vector. As the name suggests, it will store the color of each vertex of `g`. Therefore, its size must be the same as the number of vertices in `g`. The implementation is, once again, split up into *declarative* and *main iterative* part.

```
/*declarative part*/
coloredSCCs = 0; //assuming 0 = uncolored
auto coloredSCCsView = coloredSCCs.getView();

using DeviceType = typename GraphType::DeviceType;
using IndexType = typename GraphType::IndexType;
using matrixType = typename GraphType::MatrixType;

auto size = g.getNodeCount();
auto& adjMatrix = g.getAdjacencyMatrix();
//pre-fetch the matrix transposition
matrixType transposedMatrix;
transposedMatrix.getTransposition( adjMatrix );

IndexType pivot;
IndexType currColor = 1;
OutType pred( size );
auto predView = pred.getView();
OutType desc( size );
auto descView = desc.getView();
```

As there is little preparation to be done prior to the iterative step, we simply fetch template types and use them to declare variables needed, namely the vectors `pred` and `desc` for storing predecessors and descendants reachable from a given `pivot` vertex, respectively. We also initialize our color `currColor` to be equal to 1, and *uncolor* the result vector by setting it to 0.

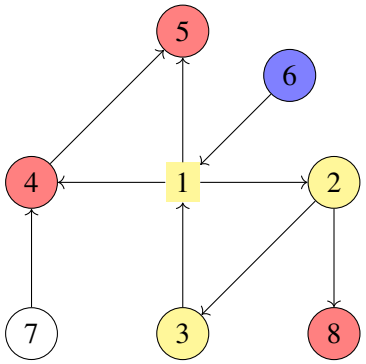
Lastly, we pre-fetch the adjacency matrix representation of `g` in `adjMatrix`, as well as transposition in `transposedMatrix`. Finding the transposition of `adjMatrix` is equal to finding the transpose graph (see section 3.3) of `g`.

```
/*main iterative part*/
while( ! TNL::all( coloredSCCsView ) ) {
    pivot = 0;
    pred = 0;
    desc = 0;

    //selecting a pivot - find the highest ID uncolored vertex via reduction
    auto selectPivot = [ = ] __cuda_callable__( IndexType vertexIdx ) -> IndexType
    {
        return ( coloredSCCsView[ vertexIdx ] ? -1 : vertexIdx );
    };
    pivot = static_cast< IndexType >(TNL::Algorithms::reduce< DeviceType >( 0,
                                                                            size,
                                                                            selectPivot,
                                                                            TNL::Max{}));

    //finding predecessors and descendants (using BFS)
    TNL::Graphs::breadthFirstSearchTransposed( adjMatrix, pivot, desc );
```

```
19      TNL::Graphs::breadthFirstSearchTransposed( transposedMatrix, pivot, pred );
20
21
22      //finding SCC = pred && desc
23      auto identifySCC = [ = ] __cuda_callable__( IndexType vertexIdx ) mutable
24      {
25          //shift by +1 so that the pivot becomes 1 and is also casted as true
26          coloredSCCsView[ vertexIdx ] =
27              ( descView[ vertexIdx ] + 1 && predView[ vertexIdx ] + 1 )
28              ? currColor : coloredSCCsView[ vertexIdx ];
29      };
30      TNL::Algorithms::parallelFor< DeviceType >( 0, size, identifySCC );
31
32      //increase color for the next iteration
33      currColor++;
34  }
```

In the main part of the implementation, we start by resetting `pivot`, `pred`, and `desc`. Then we simply find an uncolored vertex to become our new `pivot` via a parallel reduction over all vertices (lines 8 - 15). We explore all descendants and predecessors using breadth-first-search on `adjMatrix` and `transposedMatrix`, respectively (lines 18 - 19). Then, by coloring vertices found in both (including `pivot`), we mark a new strongly connected component, as seen in figure 3.3.

Figure 3.3: Choosing 1 as our **pivot**, we color vertices reachable from 1 in red, those from which we can reach 1 in blue, and those who satisfy both in yellow ⇒ we have identified an SCC {**1**, **2**, **3**}.



Lastly, we increase the color `currColor` and repeat the iterative step.

Algorithm terminates, once all vertices are colored. Since we find one component per iteration, we know the iterative step will be performed exactly k times, where k is the number of strongly connected components of `g`.

Chapter 4

Maximal Independent Set

4.1 Theory

Suppose an undirected graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$. Independent set **IS** is defined as a subset of **V** in which no two vertices are adjacent. In other words

$$\mathbf{IS} \subseteq \mathbf{V} \text{ where } \forall u, v \in \mathbf{IS} \mid \neg(u \sim v)$$

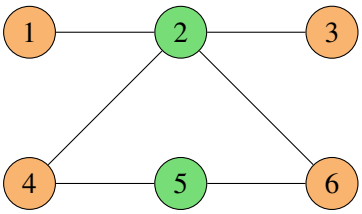
Requiring that no other vertices from **V** can be added to **IS** without breaking the adjacency rule defines a maximal independent set **MIS**. Note that multiple subsets, for which this definition holds, can exist in **V** and based on the selection of vertices, their size might vary. This makes *maximal* a term relative to the selection of vertices, not to their count (see figure 4.1). To prove whether any given set $\mathbf{MIS} \subseteq \mathbf{V}$ is a valid maximal independent set, we simply need to verify the following:

$$\forall v \in \mathbf{V} \setminus \mathbf{MIS} \text{ there } \exists u \in \mathbf{MIS} \text{ such that } (u \sim v)$$

$$\forall v \in \mathbf{MIS} \text{ there } \nexists u \in \mathbf{MIS} \text{ such that } (u \sim v)$$

Note that verifying whether a given **MIS** is the largest possible one in terms of cardinality (vertex count) is an NP-hard problem [LLRK80] and would require us finding all maximal independent sets on a given graph.

Figure 4.1: Color coding two valid **maximal independent sets** of different sizes.



4.2 Problem definition

Find a maximal independent set on an undirected unweighted graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$.

4.3 Sequential algorithm

To find a maximal independent sequentially, a simple algorithm can be used:

Algorithm 5 Sequential MIS
Input: undirected graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$
Output: boolean vector **MIS**

```
1: Initialize MIS = ∅
2: while V ≠ ∅ do
3:     choose a vertex n ∈ V
4:     add n to MIS
5:     remove n and its neighbours from V
6: end while
7: return MIS
```

This is a very intuitive approach. By removing the selected vertex along with its neighbours each iteration, we simply must end up with a set of non-adjacent vertices in the end, which satisfy our definition of a maximal independent set. Our parallel implementation hinges on the same idea.

4.4 Parallel algorithm

For finding maximal independent set of a graph, we propose a variant of **Luby's algorithm**. The basis of this algorithm were proposed by Michael Luby in 1986 [Lub86]. It can be described as follows:

Algorithm 6 Luby
Input: undirected graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$
Output: boolean vector **MIS**

```
1: Initialize MIS = ∅
2: while V ≠ ∅ do
3:     Select random subset S ⊆ V, any n ∈ V is selected with the probability 1/(2d(n))
4:     for all {u, v} ∈ E (in parallel) do
5:         if both endpoints are in S then
6:             Remove endpoints with lesser d and break ties arbitrarily
7:         end if
8:     end for
9:     MIS = MIS ∪ S
10:    Remove S and its neighbours from V
11: end while
12: return MIS
```

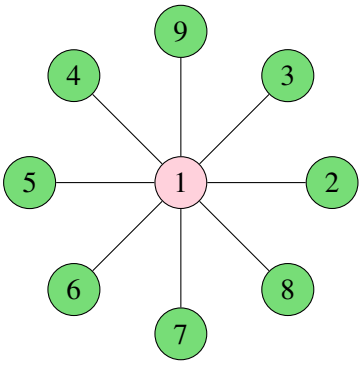
Luby's Algorithm iteratively builds a maximal independent set on graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ by selecting random subsets of vertices, while favouring those with fewer neighbours. This is reflected by choosing the probability of any given vertex n being selected to be $\frac{1}{2d(n)}$, where *d(n)* is the degree of n. The idea behind this specific selection bias is that by selecting vertices with fewer neighbours we:

1. Have a lower chance of choosing neighbouring vertices, as they are adjacent to a smaller number of vertices

2. Would like to maximize the size of our final **MIS**. By selecting vertices with less neighbours, we eliminate less vertices for future iterations

A good example to demonstrate this on is a *star* graph. Star is a graph, where all vertices are adjacent to one *root* vertex (see figure 4.2). Based on our initial selection, the size of our **MIS** can be either 1 or $(n-1)$ (assuming n vertices form the star). Using our bias here results in all vertices except the root having a $\frac{1}{2}$ chance of being selected, while the *root* vertex is selected with a probability of only $\frac{1}{2n-2}$.

Figure 4.2: Star graph with color coded **MISs**. Choosing root vertex **1** leads to a small **MIS** (pink), while preferring vertices with less neighbours **2-9** leads to a large **MIS** (green).



To prevent dividing by zero, we automatically add all solitary vertices to the maximal independent set, as they will always end up there anyway.

Once a subset **S** is selected, if we selected any two vertices adjacent to each another, we remove the one with lesser degree. When ties occur, we enforce an arbitrary rule - in our case, removing the lower index vertex.

After the removal of neighbouring vertices, we have an independent set **S** which we add to our final set **MIS**. We then remove **S** and its neighbours from the selection of vertices for next iteration.

Algorithm terminates once there are no more viable vertices left. As a result, we return a boolean vector **MIS**, which serves the role of a mask over the vertices of **G**, marking those present in the maximal independent set.

4.5 TNL implementation

We implement Luby's algorithm as a function `TNL::Graphs::maximalIndependentSet` (also referred to as `MIS_opt3` in subsections 7.5.3 and 7.6.2) defined with the following parameters:

```
1 template< typename GraphType, typename VectorType >
2 void maximalIndependentSet(const GraphType& g, VectorType& mis){/*code*/}
```

where `g` is an undirected graph, and `mis` is the vector to which our resulting set will be saved. Each element of `mis` corresponds to a vertex of `g`, with each vertex belonging to a maximal independent set being marked. Implementation can be divided into two parts - *declarative* and *main iterative*.

```
/*declarative part*/
//fetching type info
using ValueType = typename GraphType::ValueType;
using DeviceType = typename GraphType::DeviceType;
using IndexType = typename GraphType::IndexType;
using IndexVectorType = TNL::Containers::Vector< IndexType, DeviceType, IndexType >;
using BooleanVectorType = TNL::Containers::Vector< bool, DeviceType, IndexType >;

auto size = g.getNodeCount();

//fetching views of input variables
auto& adjMatrix = g.getAdjacencyMatrix();
auto adjMatrixView = adjMatrix.getConstView();
auto misView = mis.getView();

BooleanVectorType canBePicked( size );          //mask to track "pickable" vertices
auto canBePickedView = canBePicked.getView();
BooleanVectorType s( size );                    //mask to track selected vertices
auto sView = s.getView();
IndexVectorType degs( size );                   //degrees of vertices
auto degsView = degs.getView();

//# of edges - 2 counts per edge (each direction = 1)
auto totalEdges = adjMatrix.getSegments().getStorageSize();

IndexVectorType rowCapacities( size );          //# of neighbours of each vertex
adjMatrix.getRowCapacities( rowCapacities );    //fetch the capacities

//vector holding indices partitioning the list of all edges
//so that each vertex has its own partition
IndexVectorType indices( size );
indices = rowCapacities;

//compute starting indices
TNL::Algorithms::inplaceExclusiveScan( indices );
auto indicesView = indices.getView();

//vectors to store edges
IndexVectorType fromVector( totalEdges );
auto fromVectorView = fromVector.getView();
IndexVectorType toVector( totalEdges );
auto toVectorView = toVector.getView();
```

We first get type information from templates. We use these fetched types to correctly declare variables needed for the algorithm, namely:

- `canBePicked` - boolean mask for tracking vertices, that can still be picked
- `s` - boolean mask for selecting vertices in current iteration
- `degs` - vector maintaining degree (number of adjacent vertices) of each vertex

We also declare variables needed for fetching all the edges of the graph (see section 2.5 and figure 2.3).

```
//initialize, get degrees + fetch edges
auto init = [ = ] __cuda_callable__( IndexType rowIdx ) mutable
{
```

```
    //calculate degrees
    auto const row = adjMatrixView.getRow( rowIdx );
    degsView[ rowIdx ] += row.getSize();

    //add any vertex with degree 0 to mis
    misView[ rowIdx ] = ! degsView[ rowIdx ];
    canBePickedView[ rowIdx ] = degsView[ rowIdx ];

    //get edges
    IndexType starting_index = indicesView[ rowIdx ];

    for( IndexType i = 0; i < row.getSize(); i++ ) {
        fromVectorView[ starting_index + i ] = rowIdx;
        toVectorView[ starting_index + i ] = row.getColumnIndex( i );
    }
    //init variables
    sView[ rowIdx ] = false;
};
TNL::Algorithms::parallelFor< DeviceType >( 0, size, getDegrees );
```

Since we need to calculate `degs` only once, we do so here in the `init` lambda function. We also use this function to immediately add solitary vertices to our final `mis` vector, as they must end up there eventually. Lastly, we use `init` to fetch all graph edges.

```
/*main iterative part*/
while( TNL::any( canBePickedView ) ) {
    //calculate, how many vertices can be picked -> switch approach based on that
    auto fetchVertices = [ = ] __cuda_callable__( IndexType idx ) -> IndexType
    {
        return canBePickedView[ idx ] ? 1 : 0;
    };
    auto sumCanBePicked = TNL::Algorithms::reduce< DeviceType >(0,
                                                                size,
                                                                fetchVertices,
                                                                TNL::Plus{});
    bool lessThanHalfCanBePicked = sumCanBePicked < (size * 0.5);

    sView = s.getView();
    if( lessThanHalfCanBePicked ) {
        auto populateS = [ = ] __cuda_callable__( IndexType idx ) mutable
        {
            sView[ idx ] = canBePickedView[ idx ];
        };
        TNL::Algorithms::parallelFor< DeviceType >( 0, size, populateS );
    }
    else {
        do {
            auto populateS = [ = ] __cuda_callable__( IndexType idx ) mutable
        {
            if( canBePickedView[ idx ] ) {
            double rNum = randDoubleGen( idx, size );
            double prob = 1.0 / ( 2.0 * degsView[ idx ] );
            sView[ idx ] = ( rNum < ( prob ) );
            }
        };
        TNL::Algorithms::parallelFor< DeviceType >( 0, size, populateS );
```

```
        } while( sView == false );
    }
```

In the main part of our implementation, we start by selecting a subset of available vertices and store it in `s`. We employ adaptive approach to selecting vertices - **selecting a random subset** or **selecting all valid vertices**. We do this based on how many valid vertices remain. This prevents waiting for vertices to get randomly selected, when less than half remain. Note that opting for always choosing all available vertices is a valid approach. This makes the algorithm deterministic, however, meaning that running the algorithm on a given graph will always result in the same `mis`. This maximal independent set is referred to as the *lexicographically first* MIS. We believe this to be undesirable, and so we combine this approach with the random selection to retain the ability to produce different maximal independent sets. Switching the approach after less than half of the vertices remain speeds up the algorithm substantially.

```
    auto updateSetS = [ = ] __cuda_callable__( IndexType edgeIdx ) mutable
    {
        // For each edge (u, v)
        IndexType u = fromVectorView[ edgeIdx ];
        IndexType v = toVectorView[ edgeIdx ];

        // Check if both endpoints are in set S
        if( sView[ u ] && sView[ v ] ) {
            IndexType degreeU = degsView[ u ];
            IndexType degreeV = degsView[ v ];

            // Remove the vertex of lower degree from S (or arbitrarily if degrees are equal)
            if( degreeU < degreeV ) {
               sView[ u ] = false;  // Remove u
            }
            else if( degreeV < degreeU ) {
               sView[ v ] = false;  // Remove v
            }
            else {
               sView[ u < v ? u : v ] = false;
            }
        }
    };
    TNL::Algorithms::parallelFor< DeviceType >( 0, totalEdges, updateSetS );

    auto fetchN = [ = ] __cuda_callable__( IndexType edgeIdx ) mutable
    {
        IndexType u = fromVectorView[ edgeIdx ];
        IndexType v = toVectorView[ edgeIdx ];
        if( sView[ u ] ) {
            misView[ u ] = true;
            canBePickedView[ u ] = false;
            canBePickedView[ v ] = false;
        }
    };
    TNL::Algorithms::parallelFor< DeviceType >( 0, totalEdges, fetchN );
    s = false;
}
```

After selecting `s`, we go over all edges in parallel in `updateSetS` lambda and check if both endpoints were picked. If so, we proceed according to the algorithm and break ties based on vertex index.

Lastly, in `fetchN`, we go over all edges in parallel and add each seleted vertex to our final set. We then flag it and all its neighbours in `canBePicked` to not select them in future iterations. Algorithm terminates once there are no valid vertices to select anymore.

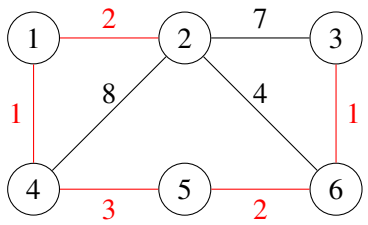
Chapter 5

Minimal Spanning Tree

5.1 Theory

Suppose a weighted undirected graph $\mathbf{G} = (\mathbf{V}, \mathbf{E}, w : \mathbf{E} \to \mathbb{N}^+)$. Minimal Spanning tree **MST** is a spanning tree on **G** that has the smallest possible edge weight sum **W**. In case of **G** being disconnected, a set of minimal spanning trees $\mathbf{MSF} = \{\mathbf{T_1}, \mathbf{T_2}, \ldots, \mathbf{T_n}\}$ (one per component) called minimal spanning forest is defined instead (see figure 5.1).

Figure 5.1: Color coding **minimal spanning tree** of a graph.



5.2 Problem definition

Find a minimal spanning tree (or forest) on an undirected weighted graph $\mathbf{G} = (\mathbf{V}, \mathbf{E}, w : \mathbf{E} \to \mathbb{N}^+)$ with unique weights.

5.3 Sequential algorithm

To find minimal spanning tree or forest sequentially, we can use a plethora of well-known algorithms, namely **Borůvka's algortihm** [Bor26], **Jarník's algorithm** (also known as **Prim's algorithm** [Pri57]), and **Kruskal's algorithm** [Kru56]. Let us show Kruskal's algorithm, which is essentially a greedy algorithm with specific ordering of edges:

Algorithm 7 Sequential MST
Input: undirected weighted graph $\mathbf{G} = (\mathbf{V}, \mathbf{E},\ w : \mathbf{E} \to \mathbb{N}^+)$
Output: minimal spanning tree/forest $\mathbf{MST} = (\mathbf{V}', \mathbf{E}',\ w : \mathbf{E}' \to \mathbb{N}^+)$

initialize $\mathbf{MST} = (\mathbf{V}' = \emptyset, \mathbf{E}' = \emptyset,\ w)$
re-order edges $\mathbf{E}$ so that $w(e_0) \leq w(e_1) \leq w(e_2) \leq \cdots \leq w(e_n)$ (ascending order by weight)
for all edges $e = \{u, v\} \in \mathbf{E}$ (ordered) **do**
    if adding edge $\{u, v\}$ to **MST** does not create any cycles **then**
        add u, v to $\mathbf{V}'$ (if they are not there already) and the edge $\{u, v\}$ to $\mathbf{E}'$
    end if
    if $\mathbf{V} = \mathbf{V}'$ **then**
        return MST
    end if
end for
return MST

The algorithm works because of the ordering of edges we chose. Note that we could run this algorithm with any ordering of edges and would still find a spanning tree or forest, but only the ascending order ensures finding the minimal spanning tree or forest in the end.

5.4 Parallel algorithm

To find minimal spanning tree, we use a variant of **Awerbuch-Shiloach algorithm** [AS87] proposed in [BKS22], which we have modified for TNL and our needs.

The main idea behind this algorithm is to initially view the graph as a collection of rooted trees (similar to section 2.4). If these trees are stars (ie. of height at most 2), a minimal edge going out of each star is found and propagated to its root. The root is then hooked based on this edge to the parent of its endpoint outside of root star. After that, we check for potential loops and break them. Then we add edges found in this iteration to the minimal spanning tree (between their original endpoints!) and the edge weight sum **W**. Lastly, we shortcut the trees, so that they can become stars and be processed in future iterations. Algorithm ends once the trees (represented in parent vector **p**) *collapse* into stars and stop changing, at which point our MST is finished. If the graph is disconnected, we end up with a minimal spanning forest MSF, where each tree is represented by a star in the final **p** vector. The algorithm can be described as follows:

Algorithm 8 TNL-adapted MST

Input: undirected weighted graph $\mathbf{G} = (\mathbf{V}, \mathbf{E},\ w : \mathbf{E} \to \mathbb{N}^+)$ (with unique weights)

Output: minimal spanning tree/forest $\mathbf{MST} = (\mathbf{V}', \mathbf{E}',\ w : \mathbf{E}' \to \mathbb{N}^+)$, sum of edge weights $\mathbf{W}$

assume $\forall n \in \mathbf{V}$ to be a rooted tree $\to \mathbf{p_n} = \mathbf{n}$
initialize vectors to track minimal outgoing edges from vertex/star **minV**, **minS**
initialize $\mathbf{MST} = (\mathbf{V}' = \emptyset, \mathbf{E}' = \emptyset,\ w)$
initialize edge weight sum $\mathbf{W} = \mathbf{0}$
while p keeps changing **do**
    for all $n \in \mathbf{V}$ (in parallel) **do**
        if n is a part of star **then**
            for all neighbours of n that are **NOT** in the same star **do**
                find edge $e = (\mathbf{n}, \mathbf{neighbour}, \mathbf{w})$ with the smallest weight **w**
                $\mathbf{minV_n} = e$
            end for
        end if
    end for
    for all $n \in \mathbf{V}$ (in parallel) **do**
        if n is a root of a star **then**
            for all children of n **do**
                find the smallest edge e among the children (if it exists)
                $\mathbf{minS_n} = e$
            end for
        end if
    end for
    for all $n \in \mathbf{V}$ (in parallel) **do**
        if n is a root of a star **AND** $\mathbf{minS_n} \neq \emptyset$ **then**
            hook n with the parent of **neighbour** of the $\mathbf{minS_n}$ edge
            $\mathbf{p_n} = \mathbf{p_{neighbour}}$
        end if
    end for
    for all $n \in \mathbf{V}$ (in parallel) **do**
        if n is a root of a star **AND** $\mathbf{n} < \mathbf{p_n}$ **AND** $\mathbf{n} = \mathbf{p_{p_n}}$ **then**
            $\mathbf{p_n} = \mathbf{n}$
        end if
    end for
    for all $n \in \mathbf{V}$ (in parallel) **do**
        if $\mathbf{minS_n} \neq \emptyset$ **AND** $\mathbf{p_n} = \mathbf{p_{neighbour}}$ **then**
            add edge in $\mathbf{minS_n}$ to the **MST**
            add edge weight **w** of edge in $\mathbf{minS_n}$ to edge weight sum **W**
        end if
    end for
    reset **misV**, **misS**
end while
return MST and **W**

Now, let us go over the algorithms in greater detail.

First we set up our parent vector **p** with the assumption, that each vertex starts as its own parent - star root (line 1). We then initialize our edge weight sum **W**, minimal spanning tree **MST**, and some helper vectors to track minimal outgoing edges per vertex and per star root (lines 2 - 4). After that, the main iteration step follows.

First thing we do each iteration is identify the smallest *outgoing edge* per each vertex inside of a star (lines 6 - 13). By outgoing edge, we mean an edge which has 1 endpoint inside of a star and 1 endpoint outside of it. So for any vertex n in a star, we look for the smallest weighted edge $\{n, x\}$, where x is not part of the same star as n. In practice, this can be checked by going over all neighbours of n and checking their parents in the **p** vector - it must not be the same as the parent of n in order to be a suitable edge. Note that a suitable edge does not have to exist - n can have no edges going outside of the star.

Next we propagate the smallest of our newly found minimal edges to the roots of their stars (lines 14 - 21). For each star root vertex, we go through its children and find the smallest outgoing edge amongst them. Once again, note that there may be no suitable edge.

Assuming there were suitable edges this iteration, we use them to *hook* the star root vertices (lines 22 - 27). By hooking, we mean changing the parent of the root in **p** vector, so that it now shares parent with the endpoint of the smallest weighted outgoing edge. So if a star root n has a child vertex m with the smallest weighted outgoing edge $\{m, x\}$ of the star rooted in n, then $\mathbf{p_n} = \mathbf{p_x}$.

After hooking, we check for cycles using the **p** vector and if we find some, we break them (lines 28 - 32).

Lastly, we add the minimal outgoing edge of each star to the minimal spanning tree, provided it did not create a cycle (lines 33 - 38). We also reset vectors used for tracking potential minimal edges to add to MST each iteration (line 39).

5.5 TNL implementation

First of all, let us state the following - unfortunately, we were unable to create a fully functional TNL-adapted implementation the way we set out to. We have created multiple different seemingly working implementations, but would later always discover that they can break and enter an infinite runtime loop. This was frustrating, as all implementations seemed reliable for smaller graphs and would even pass unit tests, but for larger graph sizes, they became unreliable and prone to breaking. Even after debugging and trying to the best of our abilities, we were unable to remove the cause of this behaviour, which happens seemingly at random, but is certainly more likely as the number of vertices and edges increase.

While this is unfortunate, we can still at least provide 2 different implementations of the algorithm, which work the best. From what we tested, the first one - `tnlMSF1` - seems to always give a correct result, provided it actually finishes. Second implementation - `tnlMSF2` - uses looping over edges and is perhaps more readable because of it, but it was introduced significantly later in the implementation process. It could therefore not be 100% correct in its results, when it finishes. It also uses some CUDA atomic operations, so it would have to be tweaked to work on CPU. Due to the reasons mentioned above, we will omit this graph problem from the following chapters. We provide the following implementations for the completeness of our work and to show current progress made.

Starting with `tnlMSF1`, it is defined with the following parameters:

```
template<
    typename inGraphType,
    typename outGraphType = inGraphType,
    typename OutType = typename inGraphType::RealType
    >
void
tnlMSF1( const inGraphType& inGraph, outGraphType& outGraph, OutType& sum ){/*code*/}
```

where `inGraph` is the graph we are finding minimal spanning tree on, `outGraph` is the graph we are going to store the tree into, and `sum` is the sum of edge weights of the minimal spanning tree. Code itself can be split into 2 main parts - *initialization* and *main iterative part*.

```
/*init part*/
using DeviceType = typename inGraphType::DeviceType;
using IndexType = typename inGraphType::IndexType;
using RealType = typename inGraphType::MatrixType::RealType;
using IndexVectorType = TNL::Containers::Vector< IndexType, DeviceType, IndexType >;
using RealVectorType = TNL::Containers::Vector< RealType, DeviceType, IndexType >;
using BoolVectorType = TNL::Containers::Vector< bool, DeviceType, IndexType >;

//to be sure
sum = 0;
RealVectorType sumV( 1, 0 );

auto& adjMatrix = inGraph.getAdjacencyMatrix();
auto adjMatrixView = adjMatrix.getConstView();
auto const size = inGraph.getNodeCount();

//setting up the outgoing graph
outGraphType tempGraph;
tempGraph.setNodeCount( size );
TNL::Containers::Vector< IndexType, DeviceType, IndexType > nodeCapacities( size );
inGraph.getAdjacencyMatrix().getRowCapacities( nodeCapacities );
tempGraph.setNodeCapacities( nodeCapacities );

auto& tempGraphMatrix = tempGraph.getAdjacencyMatrix();
auto tempGraphMatrixView = tempGraphMatrix.getView();

//helper vector to track # of edges added to each row
RealVectorType rowCapacitiesTracker( size, 0 );
auto rowCapacitiesTrackerView = rowCapacitiesTracker.getView();

IndexVectorType p( size );
BoolVectorType star( size, true );
BoolVectorType starRoot( size, true );

BoolVectorType T( size );
auto TView = T.getView();

auto starView = star.getView();
auto starRootView = starRoot.getView();
auto pView = p.getView();

//initialize parents -> each vertex is a star with it being its own parent
```

```
auto f = [ = ] __cuda_callable__( IndexType vertexIdx ) mutable
{
        pView[ vertexIdx ] = vertexIdx;
};
TNL::Algorithms::parallelFor< DeviceType >( 0, size, f );

IndexVectorType pOld( size, -1 );
auto pOldView = pOld.getView();

//Weights of the minimal outgoing edges (ie edges going to a different star)
RealVectorType minOutWeights( size, -1 );
RealVectorType minOutWeightsFrom( size, -1 );
IndexVectorType minOutWeightsTo( size, -1 );  //Destinations of the minimal outgoing edges

auto minOutWeightsView = minOutWeights.getView();
auto minOutWeightsFromView = minOutWeightsFrom.getView();
auto minOutWeightsToView = minOutWeightsTo.getView();

//define starCheck - routine to update the star status of all vertices
//(the vector must be reset to true prior to calling this)
auto starCheck = [ = ] __cuda_callable__( IndexType vertexIdx ) mutable
{
    bool isRoot = pView[ vertexIdx ] == vertexIdx;
    bool isDirectChild = pView[ pView[ vertexIdx ] ] == pView[ vertexIdx ];

    starView[ vertexIdx ] = isRoot || isDirectChild;

    // If a vertex is part of a star and is its own parent, it's a star root
    starRootView[ vertexIdx ] = starView[ vertexIdx ] && isRoot;
};
```

During initialization, we:

- retrieve variable types and device from the parameters
- set provided `sum` to 0, just in case
- retrieve adjacency matrix representation of the graph
- set up a temporary graph `tempGraph` for the minimal spanning tree. This is done because as far as we are aware, there is no way to change certain graph attributes, namely node capacities (ie. number of non-zero elements of each row of the adjacency matrix representation), without wiping its data. Because of this, we need to build `tempGraph` and count the new node capacities, then set them for `outGraph` and copy the `tempGraph` to it.
- declare and initialize vectors needed to track parent and star status of vertices - `p`, `star`, and `starRoot` - as well as helper vectors for keeping track of minimal outgoing edges and loops for loop breaking - `minOutWeights`, `minOutWeightsFrom`, `minOutWeightsTo`, and `T`
- Define a lambda function used to update star status of vertices - `starCheck`

After initialization, the main iterative part of the algorithm is repeated, until parent vector `p` stabilizes.

```
/*main part*/
do {
    pOldView = pView;
```

```
4
5       starView = true;
6       TNL::Algorithms::parallelFor< DeviceType >( 0, size, starCheck );
7
8       //calculate minimal outgoing edge from each vertex that is part of a star
9       auto findMinimalOutEdge = [ = ] __cuda_callable__( IndexType vertexIdx ) mutable
10      {
11          if( starView[ vertexIdx ] ) {
12              RealType minWeight = -1;
13              IndexType minTo = -1;
14              IndexType minFrom = -1;
15              RealType min = std::numeric_limits< RealType >::max();
16
17              auto const row = adjMatrixView.getRow( vertexIdx );
18              for( IndexType i = 0; i < row.getSize(); i++ ) {
19                  if( row.getValue( i ) < min
20                  && pView[ vertexIdx ] != pView[ row.getColumnIndex( i ) ]) {
21                      min = row.getValue( i );
22                      minWeight = min;
23                      minTo = row.getColumnIndex( i );
24                      minFrom = vertexIdx;
25                  }
26              }
27              minOutWeightsView[ vertexIdx ] = minWeight;
28              minOutWeightsToView[ vertexIdx ] = minTo;
29              minOutWeightsFromView[ vertexIdx ] = minFrom;
30          }
31      };
32      TNL::Algorithms::parallelFor< DeviceType >( 0, size, findMinimalOutEdge );
33
34      //propagate the min of outgoing edges in a star to its root
35      auto minPerStar = [ = ] __cuda_callable__( IndexType vertexIdx ) mutable
36      {
37          if( starRootView[ vertexIdx ] ) {
38              RealType minWeight = -1;
39              IndexType minTo = -1;
40              IndexType minFrom = -1;
41              RealType min = std::numeric_limits< RealType >::max();
42
43              for( IndexType i = 0; i < size; i++ ) {
44                  if( ( pView[ i ] == vertexIdx )
45                  && ( minOutWeightsView[ i ] < min )
46                  && ( minOutWeightsView[ i ] > -1 ) ) {
47                      min = minOutWeightsView[ i ];
48                      minWeight = min;
49                      minTo = minOutWeightsToView[ i ];
50                      minFrom = minOutWeightsFromView[ i ];
51                  }
52              }
53              minOutWeightsView[ vertexIdx ] = minWeight;
54              minOutWeightsToView[ vertexIdx ] = minTo;
55              minOutWeightsFromView[ vertexIdx ] = minFrom;
56          }
57      };
58      TNL::Algorithms::parallelFor< DeviceType >( 0, size, minPerStar );
59
60      //hook the stars using min edges
61      auto starHook = [ = ] __cuda_callable__( IndexType vertexIdx ) mutable
```

```
    {
        if( starRootView[ vertexIdx ] && minOutWeightsToView[ vertexIdx ] > -1 ) {

            pView[ vertexIdx ] = pOldView[ minOutWeightsToView[ vertexIdx ] ];
        }
    };
    TNL::Algorithms::parallelFor< DeviceType >( 0, size, starHook );

    //identify edges to remove
    //(smaller index edge of the two will be removed)
    auto breakLoops = [ = ] __cuda_callable__( IndexType vertexIdx ) mutable
    {
        if( starRootView[ vertexIdx ]
        && ( vertexIdx < pView[ vertexIdx ] )
        && ( vertexIdx == pView[ pView[ vertexIdx ] ] ) ) {
            TView[ vertexIdx ] = true;
        }
        else {
            TView[ vertexIdx ] = false;
        }
    };
    TNL::Algorithms::parallelFor< DeviceType >( 0, size, breakLoops );

    //remove edges
    auto breakLoops2 = [ = ] __cuda_callable__( IndexType vertexIdx ) mutable
    {
        if( TView[ vertexIdx ] ) {
            pView[ vertexIdx ] = vertexIdx;
        }
    };
    TNL::Algorithms::parallelFor< DeviceType >( 0, size, breakLoops2 );

    //fetch unique weights
    RealVectorType uniqueWeights( size, -1 );
    auto uniqueWeightsView = uniqueWeights.getView();
    auto fetchUnique = [ = ] __cuda_callable__( IndexType weightIdx ) mutable
    {
        if( ! TView[ weightIdx ]
        && minOutWeightsView[ weightIdx ] > -1
        && starRootView[ weightIdx ] ) {
            uniqueWeightsView[ weightIdx ] = minOutWeightsView[ weightIdx ];
        }
    };
    TNL::Algorithms::parallelFor< DeviceType >( 0, size, fetchUnique );

    //Cuda
    if constexpr( std::is_same< DeviceType, TNL::Devices::Cuda >::value ) {

        auto sumView = sumV.getView();
        auto updateTree = [ = ] __cuda_callable__( IndexType weightIdx ) mutable
        {
            if( uniqueWeightsView[ weightIdx ] > -1 ) {
                auto i = atomicAdd(
                        &rowCapacitiesTrackerView[ minOutWeightsFromView[ weightIdx ] ], 1 );

                auto j = atomicAdd(
                        &rowCapacitiesTrackerView[ minOutWeightsToView[ weightIdx ] ], 1 );

```

```
                auto rowFrom = tempGraphMatrixView.getRow( minOutWeightsFromView[ weightIdx ] );
                auto rowTo = tempGraphMatrixView.getRow( minOutWeightsToView( weightIdx ) );

                if( ! ( rowFrom.getValue( i ) ) ) {
                    atomicAdd( &sumView[ 0 ], uniqueWeightsView[ weightIdx ] );

                    rowFrom.setElement(
                            i,
                            minOutWeightsToView[ weightIdx ],
                            uniqueWeightsView[ weightIdx ] );

                    rowTo.setElement(
                            j,
                            minOutWeightsFromView[ weightIdx ],
                            uniqueWeightsView[ weightIdx ] );
                }
            }
        };
        TNL::Algorithms::parallelFor< DeviceType >( 0, size, updateTree );
        sum = sumView.getElement( 0 );
    }
    //host and seq
    else {

        for( int i = 0; i < size; i++ ) {
            if( uniqueWeightsView.getElement( i ) > -1 ) {
                auto from = minOutWeightsFromView.getElement( i );
                auto to = minOutWeightsToView.getElement( i );
                auto weight = uniqueWeightsView.getElement( i );

                if( tempGraphMatrixView.getElement( to, from ) == 0 ) {

                    rowCapacitiesTrackerView.setElement(
                                            from,
                                            rowCapacitiesTrackerView.getElement( from ) + 1 );

                    rowCapacitiesTrackerView.setElement(
                                            to,
                                            rowCapacitiesTrackerView.getElement( to ) + 1 );

                  sum += weight;
                  tempGraphMatrixView.setElement( from, to, weight );
                  tempGraphMatrixView.setElement( to, from, weight );
                }
            }
        }
    }

    //starCheck
    starView = true;
    TNL::Algorithms::parallelFor< DeviceType >( 0, size, starCheck );

    auto shortCut = [ = ] __cuda_callable__( IndexType vertexIdx ) mutable
    {
        if( ! starView[ vertexIdx ] ) {
            pView[ vertexIdx ] = pView[ pView[ vertexIdx ] ];
        }
    };
```

```
178     TNL::Algorithms::parallelFor< DeviceType >( 0, size, shortCut );
179
180 } while( pView != pOldView );
181
182 //update outgraph
183 outGraph.setNodeCapacities( rowCapacitiesTracker );
184 outGraph = tempGraph;
```

In the main part of the implementation, we follow the algorithm description by first updating star status of vertices, then finding the minimal outgoing edge from each vertex that is in a star. This is done in the lambda function `findMinimalOutEdge` (lines 9 - 32). Note the sequential loop inside - looping over the edges removes the need for it, as can be seen later in `tnlMSF2`. Very similar process happens in the `minPerStar` lambda (lines 35 - 58), where we propagate the smallest outgoing edge of each star to its root.

These edges are then used to hook the stars (lines 61 - 68). By hooking, we mean updating star root vertex parent to be the same as the parent of the endpoint vertex of the smallest outgoing edge from that star (line 65).

After hooking, we need to verify that no loops were created by it. We first try to identify loops and then break them (lines 72 - 92). A loop is found by simply checking, whether a star root vertex is also its grandparent - this would mean we hooked it onto a vertex whose parent is the star root vertex itself. Since we only hook by star roots, we do not need to check any other vertices.

After breaking loops, we note edges used for hooking (lines 95 - 105) and add them to our minimal spanning tree. This part of implementation is different based on device (lines 108 - 140 for GPU, and 142 - 166 for CPU). We still perform same things in both of them:

- identify endpoints of the edge and update their respective row capacities. These will be then used to set node capacities of the `outGraph`.
- add the edge itself to `tempGraph`
- add the edge weight to `sum`

Lastly, we update star status of vertices and shortcut those that are not a part of a star, so that they can become part of a star in future iterations (lines 169 - 178).

If vector `p` stopped changing, we are done and just set the appropriate node capacities of `outGraph`, then just copy `tempGraph` into it (lines 183 - 184).

Second implementation - `tnlMSF2` - is declared as:

```
1 template<
2     typename inGraphType,
3     typename outGraphType = inGraphType,
4     typename OutType = typename inGraphType::RealType
5     >
6 void
7 tnlMSF2( const inGraphType& inGraph, outGraphType& outGraph, OutType& sum ) {/*code*/}
```

where all the parameters are the same as for `tnlMSF1`. However, this implementation has 3 main parts - *initialization*, *main iterative part*, and *final tree building*.

```
/*initialization*/
using DeviceType = typename inGraphType::DeviceType;
using IndexType = typename inGraphType::IndexType;
using RealType = typename inGraphType::MatrixType::RealType;
using IndexVectorType = TNL::Containers::Vector< IndexType, DeviceType, IndexType >;
using RealVectorType = TNL::Containers::Vector< RealType, DeviceType, IndexType >;
using BoolVectorType = TNL::Containers::Vector< bool, DeviceType, IndexType >;

sum = 0;

auto& adjMatrix = inGraph.getAdjacencyMatrix();
auto adjMatrixView = adjMatrix.getConstView();
auto const size = inGraph.getNodeCount();

//internal graph setup
outGraphType tempGraph;
tempGraph.setNodeCount( size );
TNL::Containers::Vector< IndexType, DeviceType, IndexType > nodeCapacities( size );
inGraph.getAdjacencyMatrix().getRowCapacities( nodeCapacities );
tempGraph.setNodeCapacities( nodeCapacities );

auto& tempGraphMatrix = tempGraph.getAdjacencyMatrix();
auto tempGraphMatrixView = tempGraphMatrix.getView();

//needed to then correctly set capacities of the outgraph
RealVectorType rowCapacitiesTracker( size, 0 );
auto rowCapacitiesTrackerView = rowCapacitiesTracker.getView();

//edge fetch setup
auto totalEdges = adjMatrix.getSegments().getStorageSize();
IndexVectorType indices( size, -1 );
indices = nodeCapacities;
TNL::Algorithms::inplaceExclusiveScan( indices );
auto const indicesView = indices.getConstView();

IndexVectorType fromVector( totalEdges, -1 );
auto fromVectorView = fromVector.getView();
IndexVectorType toVector( totalEdges, -1 );
auto toVectorView = toVector.getView();
RealVectorType weightVector( totalEdges, -1 );
auto weightVectorView = weightVector.getView();

BoolVectorType edgeMask( totalEdges, false );
auto edgeMaskView = edgeMask.getView();

BoolVectorType loopMask( size, false );
auto loopMaskView = loopMask.getView();

IndexVectorType minOutEdge( size, -1 );
auto minOutEdgeView = minOutEdge.getView();
RealVectorType minOutEdgeWeight( size, std::numeric_limits< RealType >::max() );
auto minOutEdgeWeightView = minOutEdgeWeight.getView();

IndexVectorType minOutEdgeRoots( size, -1 );
auto minOutEdgeRootsView = minOutEdgeRoots.getView();
```

```
56  RealVectorType minOutEdgeWeightRoots( size, std::numeric_limits< RealType >::max() );
57  auto minOutEdgeWeightRootsView = minOutEdgeWeightRoots.getView();
58  IndexVectorType p( size );  //parent vector
59  auto pView = p.getView();
60  IndexVectorType pOld( size );
61  auto pOldView = pOld.getView();
62
63  BoolVectorType star( size, true );  //star vector
64  auto starView = star.getView();
65  BoolVectorType starRoot( size, true );  //star root vector
66  auto starRootView = starRoot.getView();
67
68  auto starCheck = [ = ] __cuda_callable__( IndexType vertexIdx ) mutable
69  {
70      bool isRoot = pView[ vertexIdx ] == vertexIdx;
71      bool isDirectChild = pView[ pView[ vertexIdx ] ] == pView[ vertexIdx ];
72
73      starView[ vertexIdx ] = isRoot || isDirectChild;
74
75      // If a vertex is part of a star and is its own parent, it's a star root
76      starRootView[ vertexIdx ] = starView[ vertexIdx ] && isRoot;
77  };
78
79  auto init = [ = ] __cuda_callable__( IndexType vertexIdx ) mutable
80  {
81      pView[ vertexIdx ] = vertexIdx;  //assume each vertex to be its own parent
82
83      //fetching edges
84      const auto row = adjMatrixView.getRow( vertexIdx );
85      auto starting_index = indicesView[ vertexIdx ];
86      for( IndexType i = 0; i < row.getSize(); i++ ) {
87          fromVectorView[ starting_index + i ] = vertexIdx;
88          toVectorView[ starting_index + i ] = row.getColumnIndex( i );
89          weightVectorView[ starting_index + i ] = row.getValue( i );
90      }
91  };
92  TNL::Algorithms::parallelFor< DeviceType >( 0, size, init );
```

Initialization is similar to that of `tnlMSF1`, with the addition of fetching graph edges to loop over. This is done the same way as for other graph problems (see section 2.5), but we also track edge weights. Identification of edges is done in the `init` lambda function (lines 79 - 92). 2 different helper vectors are initialized - `edgeMask` and `loopMask`. These are simple boolean vectors for flagging edges to add to the final spanning tree and edges that caused a loop, respectively.

```
1   /*main part*/
2   do {
3       pOldView = pView;
4
5       starView = true;
6       TNL::Algorithms::parallelFor< DeviceType >( 0, size, starCheck );
7
8       auto minPerVertex = [ = ] __cuda_callable__( IndexType edgeIdx ) mutable
9       {
10          auto u = fromVectorView[ edgeIdx ];
11          auto v = toVectorView[ edgeIdx ];
12          auto w = weightVectorView[ edgeIdx ];
13
```

```
        // u must be in a star
        // v must not be in a star OR be in a different star than u
        if( ! starView[ u ] || pView[ pView[ u ] ] == pView[ pView[ v ] ] ) {
            return;
        }
        else {
            RealType currMinWeight = atomicMin( &minOutEdgeWeightView[ u ], w );
            if( w <= currMinWeight ) {
                atomicExch( &minOutEdgeView[ u ], edgeIdx );
            }
        }
    };
    TNL::Algorithms::parallelFor< DeviceType >( 0, totalEdges, minPerVertex );

    auto minPerStar = [ = ] __cuda_callable__( IndexType vertexIdx ) mutable
    {
        if( starRootView[ vertexIdx ] ) {
            // Iterate through all vertices to propagate the minimal edge to the root
            for( IndexType u = 0; u < size; u++ ) {
                if( pView[ u ] == vertexIdx ) {
                    RealType w = minOutEdgeWeightView[ u ];
                    IndexType edgeIdx = minOutEdgeView[ u ];

                    // Use atomic operations to update the root's minimal edge
                    RealType currMinWeight = atomicMin(
                                                &minOutEdgeWeightRootsView[ vertexIdx ], w );
                    if( w <= currMinWeight ) {
                        atomicExch( &minOutEdgeRootsView[ vertexIdx ], edgeIdx );
                    }
                }
            }
        }
    };
    TNL::Algorithms::parallelFor< DeviceType >( 0, size, minPerStar );

    auto hook = [ = ] __cuda_callable__( IndexType vertexIdx ) mutable
    {
        //hook star roots with minimal outgoing edges
        if( starRootView[ vertexIdx ] && minOutEdgeRootsView[ vertexIdx ] > -1 ) {
            auto edge = minOutEdgeRootsView[ vertexIdx ];
            auto v = toVectorView[ edge ];
            pView[ vertexIdx ] = pView[ v ];
            edgeMaskView[ edge ] = true;
        }
    };
    TNL::Algorithms::parallelFor< DeviceType >( 0, size, hook );


    auto findLoops = [ = ] __cuda_callable__( IndexType vertexIdx ) mutable
    {
        loopMaskView[ vertexIdx ] =
            ( starRootView[ vertexIdx ]
            && ( vertexIdx < pView[ vertexIdx ] )
            && ( vertexIdx == pView[ pView[ vertexIdx ] ] ) );
    };
    TNL::Algorithms::parallelFor< DeviceType >( 0, size, findLoops );

    auto breakLoops = [ = ] __cuda_callable__( IndexType vertexIdx ) mutable
```

```
72      {
73          if( loopMaskView[ vertexIdx ] ) {
74              IndexType expected = pView[ vertexIdx ];
75              if( atomicCAS( &pView[ vertexIdx ], expected, vertexIdx ) == expected ) {
76                  auto edgeIdx = minOutEdgeRootsView[ vertexIdx ];
77                  if( edgeIdx != -1 ) {
78                      edgeMaskView[ edgeIdx ] = false;
79                  }
80              }
81          }
82      };
83      TNL::Algorithms::parallelFor< DeviceType >( 0, size, breakLoops );
84
85
86      starView = true;
87      TNL::Algorithms::parallelFor< DeviceType >( 0, size, starCheck );
88
89      auto shortcut = [ = ] __cuda_callable__( IndexType vertexIdx ) mutable
90      {
91          if( ! starView[ vertexIdx ] ) {
92              pView[ vertexIdx ] = pView[ pView[ vertexIdx ] ];
93          }
94      };
95      TNL::Algorithms::parallelFor< DeviceType >( 0, size, shortcut );
96
97      minOutEdgeView = -1;
98      minOutEdgeWeightView = std::numeric_limits< RealType >::max();
99      minOutEdgeRootsView = -1;
100     minOutEdgeWeightRootsView = std::numeric_limits< RealType >::max();
101
102 } while( pView != pOldView );
```

The main part of this implementation is once again similar to the first one, but it has been modified to utilize looping over edges, and the tree building has been taken out. We once start by finding a minimal outgoing edge per vertex and then per star, each in its respective lambda function (lines 8 - 47). Note the use of atomic functions, namely `atomicMin` and `atomicExch` for updating of the currently considered minimal outgoing edge.

Hooking follows (lines 49 - 59). Since we have a list of all edges on hand in this implementation, we are able to mark the edges we use for hooking, hence the `edgeMask` vector.

Next step is to find and break loops that we might have created (lines 62 - 83). We use another atomic operation - compare and swap - to break the loops (line 75) and if we do so, we simply *unmark* the edge responsible for the loop in the `edgeMask`.

Lastly, a shortcutting identical to that in `tnlMSF1` is performed, as well as a reset of some helper variables for future iterations.

```
1   /*Final tree building*/
2   RealVectorType sumV( 1, 0 );
3   auto sumVView = sumV.getView();
4
5   auto buildTree = [ = ] __cuda_callable__( IndexType edgeIdx ) mutable
6   {
```

```
    if( edgeMaskView[ edgeIdx ] ) {
        auto u = fromVectorView[ edgeIdx ];
        auto v = toVectorView[ edgeIdx ];
        auto w = weightVectorView[ edgeIdx ];

        auto i = atomicAdd( &rowCapacitiesTrackerView[ u ], 1 );
        auto j = atomicAdd( &rowCapacitiesTrackerView[ v ], 1 );

        auto rowFrom = tempGraphMatrixView.getRow( u );
        auto rowTo = tempGraphMatrixView.getRow( v );

        rowFrom.setElement( i, v, w );
        rowTo.setElement( j, u, w );

        atomicAdd( &sumVView[ 0 ], w );
    }
};
TNL::Algorithms::parallelFor< DeviceType >( 0, totalEdges, buildTree );

outGraph.setNodeCapacities( rowCapacitiesTracker );
outGraph = tempGraph;
sum = sumVView.getElement( 0 );
```

Once parent vector `p` stops changing, we assume that we are done and begin the last part - building of the tree. We do so in `buildTree` lambda function, where we simply add all edges *marked* by the `edgeMask` vector to the empty `tempGraph`. While doing this, we count individual row capacities and the edge weight sum. After this, we simply set the node capacities of `outGraph` and copy our `tempGraph` into it, ending the algorithm.

Chapter 6

Unit tests

Unit test is an additional code designed to test the latest version of implementation after making some changes. Using this code, the latest implementation of our algorithm is executed with a range of inputs and the results are then compared against expected correct values. We have created unit tests for all the algorithms mentioned in this work. In some cases, we created *general verification* functions to verify the result for any given graph. Overall, unit tests for all the algorithms were implemented, while verification functions for 2 algorithms were made.

6.1 Verification functions

For some of our algorithms, namely for finding maximal independent sets and connected components, we have created boolean functions `isMIS` and `isCC` to confirm algorithm correctness for any given graph `g`.

6.1.1 isMIS

To confirm, whether a result vector `mis` marks a maximal independent set of `g`, 2 things have to be verified:

1. If v is in **MIS**, none of its neighbours are in **MIS**. If this is not the case, **MIS** is not an independent set.
2. If v is not in **MIS**, at least one of its neighbours is in **MIS**. If this is not the case, **MIS** is not maximal.

In order to confirm this, we go over all vertices in parallel and perform checks. Our implementation is seen in the following code snippet

```
template< typename GraphType, typename VectorT >
bool
isMIS( const GraphType& g, VectorT& mis )
{
    using DeviceType = typename GraphType::DeviceType;
    using IndexType = typename GraphType::IndexType;

    auto& adjMatrix = g.getAdjacencyMatrix();
    auto size = g.getNodeCount();
    auto matrixView = adjMatrix.getConstView();
```

```
    auto misView = mis.getConstView();

    auto testNeighbours = [ = ] __cuda_callable__( IndexType idx ) -> bool
    {
        auto row = matrixView.getRow( idx );
        //if the vertex IS in MIS - cant have neighbours in MIS
        if( misView[ idx ] ) {
            for( IndexType i = 0; i < row.getSize(); i++ ) {
                if( row.getValue( i ) ) {
                    if( misView[ row.getColumnIndex( i ) ] ) {
                        return true;
                    }
                }
            }
        }
        //if NOT in MIS - must have at least one neighbour in MIS
        else {
            for( IndexType i = 0; i < row.getSize(); i++ ) {
                if( row.getValue( i ) ) {
                    if( misView[ row.getColumnIndex( i ) ] ) {
                        return false;
                    }
                }
            }
            return true;
        }
        return false;
    };

    return !TNL::Algorithms::reduce< DeviceType >(0, size, testNeighbours, TNL::LogicalOr());
}
```

6.1.2 isCC

In order to verify that a given vector `cc` represents a set of connected components of `g`, we need to check each component. We find the root of each component and run **breadth-first-search** from it to see all reachable vertices. We then compare this set to that stored in `cc`. If all components match, `cc` is valid. Implementation code is shown below

```
template< typename GraphType, typename OutType >
bool
isCC( const GraphType& g, OutType& cc )
{
    using IndexType = typename OutType::IndexType;
    using RealType = typename OutType::RealType;
    using DeviceType = typename GraphType::DeviceType;
    using BooleanVectorType=TNL::Containers::Vector< bool, DeviceType, IndexType >;
    using IndexVectorType=TNL::Containers::Vector< IndexType, DeviceType, IndexType >;

    //identify roots of components
    BooleanVectorType componentGroups( cc.getSize(), false );
    auto ccView = cc.getView();
    auto componentGroupsView = componentGroups.getView();

    //(set indices of all vertices in cc to true)
    auto identifyRoots = [ = ] __cuda_callable__( IndexType vertexIdx ) mutable
```

```
18      {
19          if( ! componentGroupsView[ ccView[ vertexIdx ] ] ) {
20              componentGroupsView[ ccView[ vertexIdx ] ] = true;
21          }
22      };
23      TNL::Algorithms::parallelFor< DeviceType >( 0, cc.getSize(), identifyRoots );
24
25      //use BFS on each root and verify the results
26      while( TNL::any( componentGroups ) ) {
27          //fetch the largest vertex root
28          auto fetchMinIndex = [ = ] __cuda_callable__( IndexType vertexIdx ) -> IndexType
29          {
30              if( componentGroupsView[ vertexIdx ] ) {
31                  return vertexIdx;
32              }
33              return -1;
34          };
35          auto root = TNL::Algorithms::reduce< DeviceType >(  0,
36                                                              cc.getSize(),
37                                                              fetchMinIndex,
38                                                              TNL::Max{});
39
40          //perform BFS
41          IndexVectorType reachableFromRoot( cc.getSize(), false );
42          TNL::Graphs::breadthFirstSearch( g, root, reachableFromRoot );
43
44          //compare
45          auto reachableFromRootView = reachableFromRoot.getView();
46          auto fetchCompare = [ = ] __cuda_callable__( IndexType vertexIdx ) -> bool
47          {
48              //if current vertex is reachable via BFS, ccView[vertexIdx] must == root
49              if( reachableFromRootView[ vertexIdx ] >= 0 ) {
50                  return ( ccView[ vertexIdx ] == root );
51              }
52              //if current vertex is NOT reachable via BFS, ccView[vertexIdx] must != root
53              return ( ccView[ vertexIdx ] != root );
54          };
55          bool isValidComponent = TNL::Algorithms::reduce< DeviceType >(  0,
56                                                                          cc.getSize(),
57                                                                          fetchCompare,
58                                                                          TNL::LogicalAnd{});
59
60          if( ! isValidComponent ) {
61              return false;
62          }
63
64          //remove current root from selection
65          auto clearRoot = [ = ] __cuda_callable__( IndexType rootIdx ) mutable
66          {
67              if( rootIdx == root ) {
68                  componentGroupsView[ rootIdx ] = false;
69              }
70          };
71          TNL::Algorithms::parallelFor< DeviceType >( 0, cc.getSize(), clearRoot );
72      }
73      //if we get here, it means no component check failed
74      return true;
75  }
```

6.2 Gtest

For implementing unit tests, we use rather well-known GoogleTest (Gtest) [Goo24] - Google's C++ testing and mocking framework. To verify our algorithms, we use `TYPED_TEST_SUITE` environment with individual `TYPED_TEST`s in it. We do this to test multiple graph and device configurations (different template types) using the same algorithm interface.

Each algorithm has its own `TYPED_TEST_SUITE` with `TYPED_TEST`s. Inside of each `TYPED_TEST`, we perform our algorithm on a test graph and compare the result to the expected result using `ASSERT_EQ`. Unit tests roughly use the following structure:

```
// other includes
#include <gtest/gtest.h>

// test fixture for typed tests
template< typename Matrix >
class GraphTest : public ::testing::Test
{
protected:
   using MatrixType = Matrix;
   using GraphType = TNL::Graphs::Graph< MatrixType >;
};

// types for which MatrixTest is instantiated
using GraphTestTypes = ::testing::Types</*Matrix types to test*/>
TYPED_TEST_SUITE( GraphTest, GraphTestTypes );

TYPED_TEST( GraphTest, test )
{
    using GraphType = typename TestFixture::GraphType;
    using DeviceType = typename GraphType::DeviceType;
    using IndexType = typename GraphType::IndexType;

    /*
    Declaring a test graph and result
    Running Algorithm
    */

   ASSERT_EQ( /*result*/, /*expected result*/ );
}
```

Chapter 7

Benchmarks

Benchmarking process is crucial when it comes to high performance code. Benchmark is an additional code designed to measure and log performance of our code. In order to assess whether some change in the code leads to more efficient algorithm, we log things such as runtime and memory usage. As part of our work, we have implemented benchmarks for all our algorithms, except the minimal spanning tree algorithm. Note that we utilized an already established graph benchmark files, present in TNL, as a baseline for ours.

7.1 Benchmarks structure

In our benchmark script, we have modified TNLs implementation of the environment. By default, user can specify the graphs to run algorithms on via `--input-file` parameter, where you load in the file containing the graph. An output file for storing the benchmark log can be specified using `--output-file`. In TNLs default configuration, each algorithm is measured **10 times per matrix storage format kernel per supported device type**. Number of runs and device can be specified using `--loops` and `--device` parameters, respectively. While TNL offers multiple kernels for multiple storage formats, in our work and all our benchmarking efforts, we focus solely on **CSR** related ones, namely **CSRScalar**, **CSRVector**, **CSRLight**, and **CSRAdaptive** kernels. This is due to 3 main reasons:

1. CSR format is considered standard
2. **CSRScalar** is the only kernel currently usable on all 3 devices
3. While some versions of our algorithms work with all storage formats supported by TNL, optimized versions seem to be unreliable when using **Ellpack** storage format kernels. To be more specific, the pre-fetching of edges and looping over them in parallel (see sections 2.5 and 4.5) sometimes lead to wrong results. At other times, it is correct and actually faster than with CSR, but reliability and correctness are of higher priority.

From this point onward, any results and conclusions are drawn from benchmarks performed using **CSRScalar**, unless stated otherwise.

When we log performance of our algorithms, we note:

- `device` - device the algorithm is performed on
- `format` - storage format kernel used

- `algorithm` - name
- `performer` - same as `device`
- `time` - time averaged from all runs of the algorithm
- `loops` - number of runs given to each algorithm
- `bandwidth` - memory bandwidth

and few other variables. For those algorithms, for which we have made a *general verification* function, results are verified after each run.

Lastly, we would like to closer specify the 3 devices we are going to perform benchmarks on - **CPU**, **HOST**, and **GPU**. By CPU, we mean sequential execution on the CPU using a single thread. By HOST, we mean parallel execution on the CPU using OpenMP configured to use 16 threads. GPU means parallel execution using a GPU (see section 7.4 for more details).

7.2 External libraries

Initially, we had plans to benchmark our implementations against other libraries, namely **Graph-BLAST** (part of **Gunrock** [WPD+17]) and **cuGraph** [AI24]). Unfortunately, we have had problems with setting them up for one reason or another. For the issues that we were unable to solve, we have submitted bug reports. This is quite unfortunate, as it would have provided a much needed comparison and framing for our results. We comment on this further in the thesis conclusion, but aside from technical difficulties, poor time management and putting higher priority on other aspects of the thesis certainly played a role in why we were unable to complete what we have set out to do in regards to external libraries. These are wrong decisions we have made, and we acknowledge them.

7.3 Benchmarking procedures

Due to unforeseen problems regarding external libraries and time constrains, we have decided to settle on the following benchmarking procedures:

- **CPU** $\times$ **GPU** - since TNL allows us to employ somewhat hardware-agnostic approach, we can directly compare running the same code sequentially, using OpenMP, and on a GPU. We can explore which graph characteristics lead to performance gains or drops. Furthermore, we can find the point at which the overhead associated with GPU execution is overshadowed by the benefit of running in parallel.
- **Optimizations** - After implementing the algorithms, we went through an iterative process of modifying them to improve their (GPU) performance and to better adapt them for TNL. We can measure speedups or slowdowns caused by these changes and comment on why that may be.

For our benchmarks, we have generated a handful of directed graphs of varying *size* and *edge density*. By size, we mean the number of vertices in a graph. By edge density, we denote the percentage of directed edges present out of all the possible ones. In other words, how close the graph is to a complete graph of the same size.

If a directed graph is provided, an undirected counterpart with all one-sided edges filled in to be bi-directional is created automatically. For algorithms performed on an undirected graph, it is then used instead. All figure graphs shown are using logarithmic scale.

For tables with times, we use the following layout:

Table for Density 0.XX							
	Time (s)			Comparison			
Size	CPU	GPU	HOST	HOST × CPU	GPU × CPU	GPU × HOST	TE

where:

- **Table for Density 0.XX** - States the edge density of graphs we measured the times on. We use 0.1, 0.45, and 0.65
- **Size** - Column stating graph size (ie. number of vertices).
- **Time** - Columns with the actual times measured. Each device has its own column.
- **Comparison** - Columns comparing devices - when comparing 2 devices $\mathbf{a} \times \mathbf{b}$, we calculate $\frac{\mathbf{b}}{\mathbf{a}}$ (ie. performance of **a** compared to **b**). Result shows how much faster or slower **a** is compared to **b** as a baseline. So if $\frac{\mathbf{b}}{\mathbf{a}} < \mathbf{1}$, **a** is slower. We use *e* notation to compact the numbers and to highlight the orders of magnitude.
- **TE** - *Thread Efficiency*. In this column, we are trying to show the efficiency of a single thread performance, when running on CPU versus on HOST. We calculate it as $\frac{\mathbf{CPU\ Time}}{\mathbf{HOST\ Time * 16}}$. The idea here is to show whether HOST, which uses 16 threads using OpenMP, manages to use them better than CPU its 1 thread. In other words, HOST must perform at least 16 times better than CPU for it to be more *thread efficient*.

7.4 Benchmark specification

For benchmarking, we have used a high performance computing cluster **HELIOS** at the Department of Mathematics, Faculty of Nuclear Sciences and Physical Engineering, Czech Technical University in Prague. Hardware information in tables 7.1 and 7.2 were filled in based on information accessible in **HELIOS** cluster documentation [Str24].

System Component	**Specification**
CPU	Intel XEON Gold 6134 @ 3.20GHz
Cores	8 cores per CPU, 8 threads total
CPU Cache	24.75 MB L3
RAM	384 GB
GPU	×

Table 7.1: System specification for the CPU benchmarks

System Component	Specification
CPU	Intel XEON Gold 6130 @ 2.10GHz
Cores	16 cores per CPU, 16 threads total
CPU Cache	22 MB L3 Cache
RAM	384 GB
GPU	4x NVIDIA Tesla V100 SXM2

Table 7.2: System specification for the GPU benchmarks

For completeness, we would like to also provide `.json` metadata files generated alongside our benchmarks, which capture relevant information about the system and benchmarking setup. For the CPU and HOST benchmarks, files contained the following relevant information;


```
{
        "CPU cache sizes (L1d, L1i, L2, L3) (kiB)": "32, 32, 1024, 25344",
        "CPU cores": "8",
        "CPU max frequency (MHz)": "3201.000000",
        "CPU model name": " Intel(R) Xeon(R) Gold 6134 CPU @ 3.20GHz",
        "CPU threads per core": "1",
        "OpenMP enabled": "yes",
        "OpenMP threads": "16",
        "architecture": "AMD64",
        "compiler": "g++ v11.3.0",
        "system": "Linux",
        "system release": "3.10.0-862.el7.x86_64"
}
```


Note that, while enabled, OpenMP was not used when performing sequential benchmarks. For the GPU, the file looks roughly like this:


```
{
        "CPU cache sizes (L1d, L1i, L2, L3) (kiB)": "32, 32, 1024, 22528",
        "CPU cores": "16",
        "CPU max frequency (MHz)": "3700.000000",
        "CPU model name": " Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz",
        "CPU threads per core": "1",
        "GPU CUDA cores": "5120",
        "GPU architecture": "7.0",
        "GPU clock rate (MHz)": "1530.000000",
        "GPU global memory (GB)": "16.928342",
        "GPU memory ECC enabled": "1",
        "GPU memory clock rate (MHz)": "877.000000",
        "GPU name": "Tesla V100-SXM2-16GB",
        "OpenMP enabled": "no",
        "OpenMP threads": "0",
        "architecture": "AMD64",
        "compiler": "Nvidia NVCC (11.7.99)",
        "system": "Linux",
        "system release": "3.10.0-1127.el7.x86_64"
}
```

7.5 CPU × GPU

For comparison of CPU and GPU performance, we have measured the performance of our algorithms when executed sequentially (CPU), in parallel on the CPU using OpenMP with 16 threads (HOST), and in parallel using CUDA (GPU). Naturally, we used the same set of graphs for all devices.

7.5.1 CC

For connected components, we have focused on 2 implementations of our algorithm:

- `CC_base` - Initial implementation, which does not utilize parallel loops over edges, but rather over vertices. This implementation is not shown here, as we believe it is inferior to the optimized one, considering our focus on the GPU execution performance (see figure 7.13). Note, however, that it should work with all storage format kernels.
- `CC_opt2` - Implementation shown in section 2.5.

For comparison between them, see section 7.6.1. Regarding their CPU × GPU performance, some trends can be seen from the results.

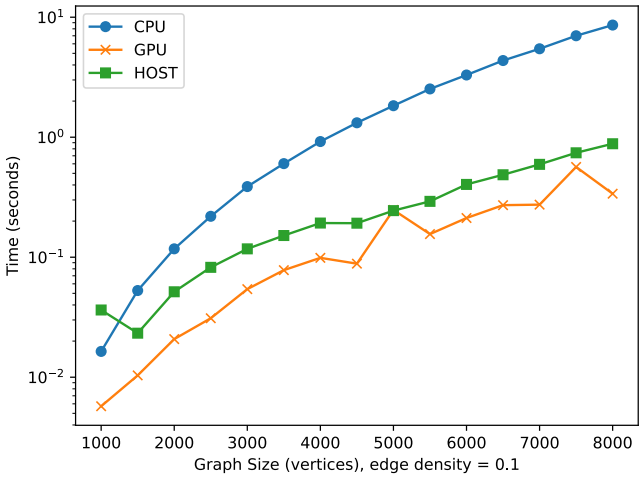
For `CC_base`, We can see that GPU performs better than sequential execution across various graph densities, while achieving various results when compared to HOST (see figures 7.1 and 7.2).

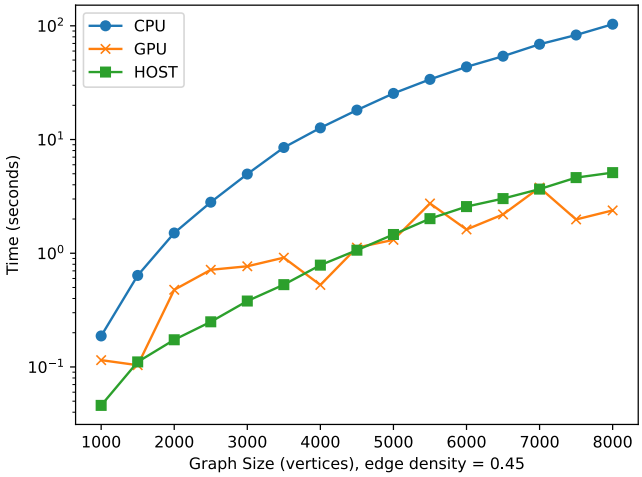
GPU being faster than sequential execution seems like a sound conclusion, as `CC_base` implementation does not iterate over edges in its *hooking* phase, but rather over vertices. This forces a sequential loop for each vertex to go through its neighbours. This has much worse impact on the CPU, because GPU (and HOST to some extend) can at least perform these sequential loops for multiple vertices at once. Sequential CPU execution is, on average, forced to perform a nested loop of size m in a loop of size n, where m is the average degree of a vertex in a given graph, and n is the size of that graph. For a worst case scenario - complete graph - hooking on a CPU would have time complexity of $\mathbf{O(n^2)}$.

From the times measured we can see, that for sparse graphs, GPU outperforms other devices. As the edge density increases, its performance becomes more and more comparable to that of HOST, While CPU execution is always the slowest one.

For `CC_opt2`, GPU clearly outperforms both CPU executions by a landslide, as can be seen in figures 7.3 and 7.4. This makes sense, as looping over edges removes the inner forced sequential loop for each vertex during *hooking* phase. While there are almost certainly more edges than vertices (ie. the *hooking* phase loop is larger now), this way all can be done fully and efficiently in parallel with no nested loops, which benefits the GPU extensively. As the edge density increases, so does the amount of edges to loop through, which in turn makes the GPU performance even more pronounced.

Figure 7.1: Comparison of `CC_base` algorithm performance on all devices on different edge densities





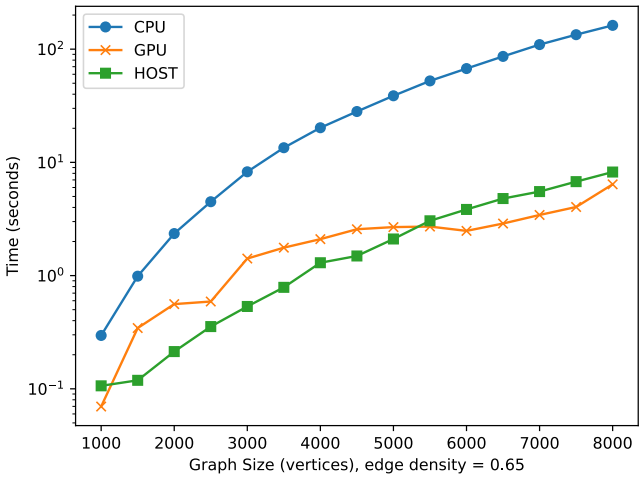


Figure 7.2: Tables for `CC_base` graphs in figure 7.1

Table for Density 0.1							
	Time (s)			Comparison			
Size	CPU	GPU	HOST	HOST × CPU	GPU × CPU	GPU × HOST	TE
1000	1.64e-02	5.73e-03	3.63e-02	4.52e-01	2.86e+00	6.34e+00	0.0282
1500	5.28e-02	1.04e-02	2.33e-02	2.27e+00	5.10e+00	2.25e+00	0.1417
2000	1.17e-01	2.08e-02	5.15e-02	2.28e+00	5.65e+00	2.48e+00	0.1425
2500	2.19e-01	3.10e-02	8.23e-02	2.66e+00	7.07e+00	2.66e+00	0.1664
3000	3.88e-01	5.42e-02	1.17e-01	3.30e+00	7.15e+00	2.17e+00	0.2064
3500	6.02e-01	7.80e-02	1.52e-01	3.97e+00	7.72e+00	1.94e+00	0.2484
4000	9.20e-01	9.88e-02	1.93e-01	4.78e+00	9.31e+00	1.95e+00	0.2986
4500	1.32e+00	8.83e-02	1.92e-01	6.87e+00	1.49e+01	2.17e+00	0.4291
5000	1.83e+00	2.48e-01	2.44e-01	7.49e+00	7.39e+00	9.86e-01	0.4679
5500	2.52e+00	1.56e-01	2.92e-01	8.65e+00	1.62e+01	1.87e+00	0.5404
6000	3.30e+00	2.12e-01	4.04e-01	8.16e+00	1.56e+01	1.91e+00	0.5102
6500	4.35e+00	2.71e-01	4.86e-01	8.95e+00	1.60e+01	1.79e+00	0.5594
7000	5.46e+00	2.74e-01	5.94e-01	9.19e+00	1.99e+01	2.17e+00	0.5741
7500	7.01e+00	5.65e-01	7.41e-01	9.45e+00	1.24e+01	1.31e+00	0.5906
8000	8.60e+00	3.38e-01	8.82e-01	9.75e+00	2.54e+01	2.61e+00	0.6093

Table for Density 0.45							
	Time (s)			Comparison			
Size	CPU	GPU	HOST	HOST × CPU	GPU × CPU	GPU × HOST	TE
1000	1.87e-01	1.15e-01	4.59e-02	4.08e+00	1.63e+00	4.00e-01	0.2550
1500	6.38e-01	1.04e-01	1.11e-01	5.76e+00	6.16e+00	1.07e+00	0.3601
2000	1.51e+00	4.77e-01	1.73e-01	8.69e+00	3.16e+00	3.64e-01	0.5431
2500	2.81e+00	7.15e-01	2.49e-01	1.13e+01	3.93e+00	3.49e-01	0.7043
3000	4.97e+00	7.68e-01	3.80e-01	1.31e+01	6.47e+00	4.95e-01	0.8176
3500	8.51e+00	9.14e-01	5.29e-01	1.61e+01	9.31e+00	5.79e-01	1.0046
4000	1.27e+01	5.26e-01	7.86e-01	1.61e+01	2.41e+01	1.49e+00	1.0071
4500	1.82e+01	1.12e+00	1.06e+00	1.71e+01	1.62e+01	9.49e-01	1.0664
5000	2.54e+01	1.31e+00	1.46e+00	1.74e+01	1.94e+01	1.11e+00	1.0880
5500	3.38e+01	2.74e+00	2.01e+00	1.68e+01	1.23e+01	7.35e-01	1.0489
6000	4.36e+01	1.62e+00	2.57e+00	1.69e+01	2.69e+01	1.59e+00	1.0585
6500	5.41e+01	2.20e+00	3.02e+00	1.79e+01	2.46e+01	1.38e+00	1.1173
7000	6.89e+01	3.78e+00	3.66e+00	1.88e+01	1.82e+01	9.70e-01	1.1743
7500	8.30e+01	1.98e+00	4.63e+00	1.79e+01	4.19e+01	2.33e+00	1.1219
8000	1.03e+02	2.38e+00	5.12e+00	2.02e+01	4.34e+01	2.15e+00	1.2618

Table for Density 0.65							
	Time (s)			Comparison			
Size	CPU	GPU	HOST	HOST × CPU	GPU × CPU	GPU × HOST	TE
1000	2.96e-01	6.98e-02	1.06e-01	2.79e+00	4.24e+00	1.52e+00	0.1741
1500	9.88e-01	3.44e-01	1.19e-01	8.31e+00	2.87e+00	3.45e-01	0.5195
2000	2.35e+00	5.60e-01	2.13e-01	1.10e+01	4.20e+00	3.81e-01	0.6891
2500	4.49e+00	5.90e-01	3.55e-01	1.27e+01	7.61e+00	6.01e-01	0.7919
3000	8.26e+00	1.41e+00	5.34e-01	1.55e+01	5.84e+00	3.77e-01	0.9667
3500	1.35e+01	1.76e+00	7.88e-01	1.71e+01	7.63e+00	4.46e-01	1.0688
4000	2.02e+01	2.10e+00	1.30e+00	1.56e+01	9.64e+00	6.19e-01	0.9731
4500	2.81e+01	2.57e+00	1.49e+00	1.89e+01	1.10e+01	5.80e-01	1.1796
5000	3.87e+01	2.68e+00	2.11e+00	1.84e+01	1.44e+01	7.85e-01	1.1498
5500	5.24e+01	2.71e+00	3.05e+00	1.72e+01	1.93e+01	1.12e+00	1.0749
6000	6.74e+01	2.48e+00	3.84e+00	1.75e+01	2.71e+01	1.55e+00	1.0968
6500	8.63e+01	2.88e+00	4.79e+00	1.80e+01	2.99e+01	1.66e+00	1.1252
7000	1.10e+02	3.43e+00	5.52e+00	1.99e+01	3.20e+01	1.61e+00	1.2419
7500	1.34e+02	4.04e+00	6.76e+00	1.99e+01	3.33e+01	1.67e+00	1.2411
8000	1.62e+02	6.40e+00	8.22e+00	1.97e+01	2.53e+01	1.28e+00	1.2337

Figure 7.3: Comparison of `CC_opt2` performance on all devices on different edge densities

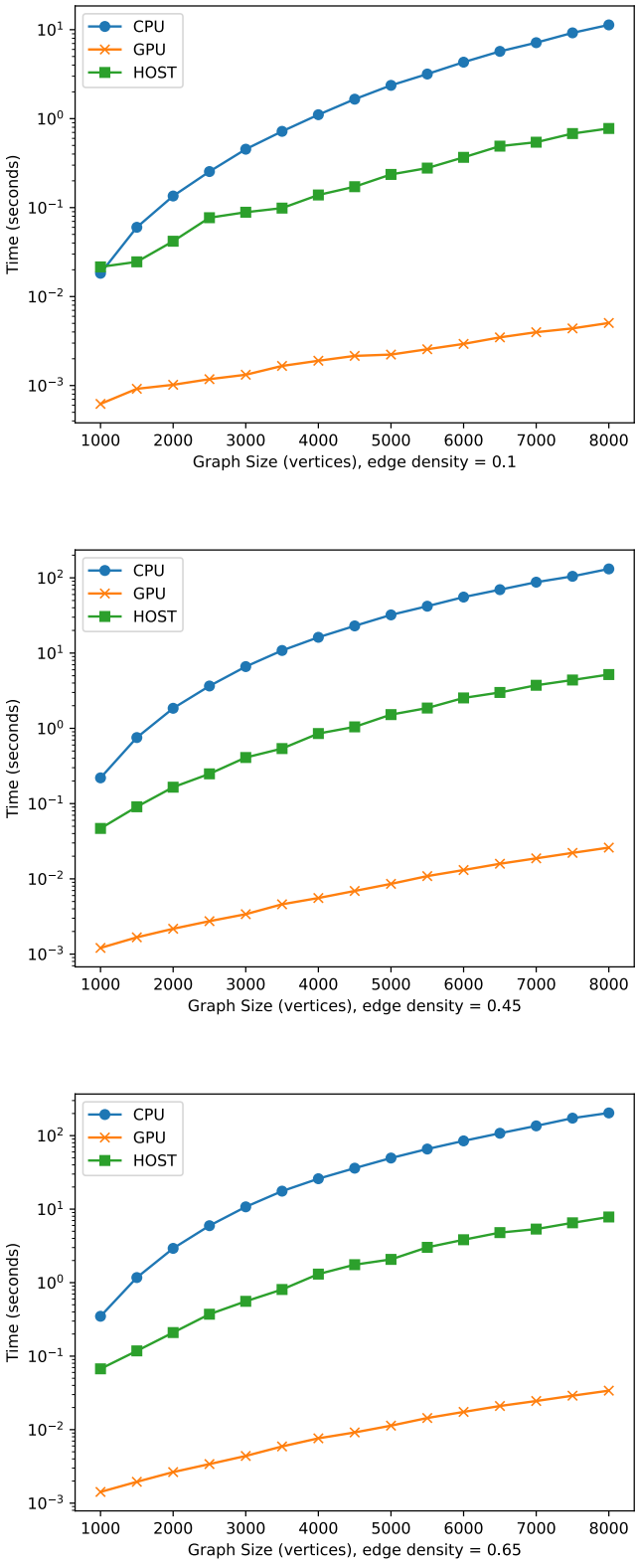


Figure 7.4: Tables for `CC_opt2` graphs in figure 7.3

Table for Density 0.1							
	Time (s)			Comparison			
Size	CPU	GPU	HOST	HOST × CPU	GPU × CPU	GPU × HOST	TE
1000	1.83e-02	6.22e-04	2.16e-02	8.48e-01	2.94e+01	3.47e+01	0.0530
1500	6.01e-02	9.18e-04	2.46e-02	2.44e+00	6.54e+01	2.68e+01	0.1528
2000	1.35e-01	1.02e-03	4.19e-02	3.22e+00	1.32e+02	4.11e+01	0.2015
2500	2.54e-01	1.18e-03	7.70e-02	3.30e+00	2.16e+02	6.53e+01	0.2064
3000	4.53e-01	1.32e-03	8.87e-02	5.11e+00	3.43e+02	6.71e+01	0.3194
3500	7.20e-01	1.66e-03	9.87e-02	7.29e+00	4.33e+02	5.95e+01	0.4556
4000	1.11e+00	1.90e-03	1.39e-01	7.99e+00	5.83e+02	7.30e+01	0.4991
4500	1.65e+00	2.15e-03	1.72e-01	9.63e+00	7.69e+02	7.98e+01	0.6021
5000	2.37e+00	2.23e-03	2.36e-01	1.00e+01	1.06e+03	1.06e+02	0.6268
5500	3.17e+00	2.56e-03	2.78e-01	1.14e+01	1.24e+03	1.09e+02	0.7131
6000	4.31e+00	2.94e-03	3.67e-01	1.17e+01	1.47e+03	1.25e+02	0.7340
6500	5.71e+00	3.48e-03	4.91e-01	1.16e+01	1.64e+03	1.41e+02	0.7265
7000	7.14e+00	3.98e-03	5.43e-01	1.31e+01	1.80e+03	1.37e+02	0.8215
7500	9.21e+00	4.39e-03	6.80e-01	1.36e+01	2.10e+03	1.55e+02	0.8469
8000	1.13e+01	5.05e-03	7.75e-01	1.46e+01	2.24e+03	1.54e+02	0.9124

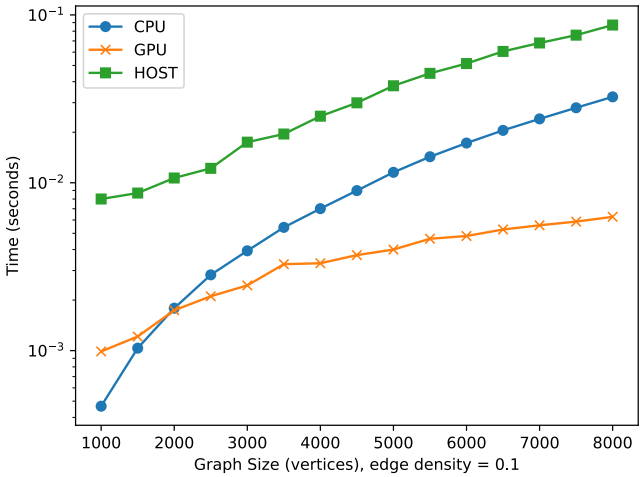
Table for Density 0.45							
	Time (s)			Comparison			
Size	CPU	GPU	HOST	HOST × CPU	GPU × CPU	GPU × HOST	TE
1000	2.20e-01	1.21e-03	4.68e-02	4.70e+00	1.82e+02	3.87e+01	0.2936
1500	7.54e-01	1.67e-03	9.08e-02	8.31e+00	4.51e+02	5.43e+01	0.5193
2000	1.85e+00	2.17e-03	1.65e-01	1.12e+01	8.50e+02	7.60e+01	0.6989
2500	3.65e+00	2.74e-03	2.49e-01	1.47e+01	1.33e+03	9.07e+01	0.9185
3000	6.61e+00	3.39e-03	4.10e-01	1.61e+01	1.95e+03	1.21e+02	1.0090
3500	1.08e+01	4.59e-03	5.39e-01	2.01e+01	2.36e+03	1.17e+02	1.2562
4000	1.62e+01	5.56e-03	8.53e-01	1.90e+01	2.91e+03	1.53e+02	1.1876
4500	2.30e+01	6.91e-03	1.05e+00	2.20e+01	3.33e+03	1.51e+02	1.3723
5000	3.22e+01	8.58e-03	1.52e+00	2.11e+01	3.75e+03	1.77e+02	1.3205
5500	4.20e+01	1.09e-02	1.86e+00	2.25e+01	3.86e+03	1.71e+02	1.4091
6000	5.55e+01	1.31e-02	2.54e+00	2.19e+01	4.24e+03	1.94e+02	1.3678
6500	6.96e+01	1.59e-02	3.00e+00	2.32e+01	4.39e+03	1.89e+02	1.4513
7000	8.75e+01	1.87e-02	3.73e+00	2.34e+01	4.67e+03	1.99e+02	1.4647
7500	1.05e+02	2.21e-02	4.39e+00	2.39e+01	4.73e+03	1.98e+02	1.4918
8000	1.31e+02	2.60e-02	5.19e+00	2.53e+01	5.05e+03	1.99e+02	1.5821

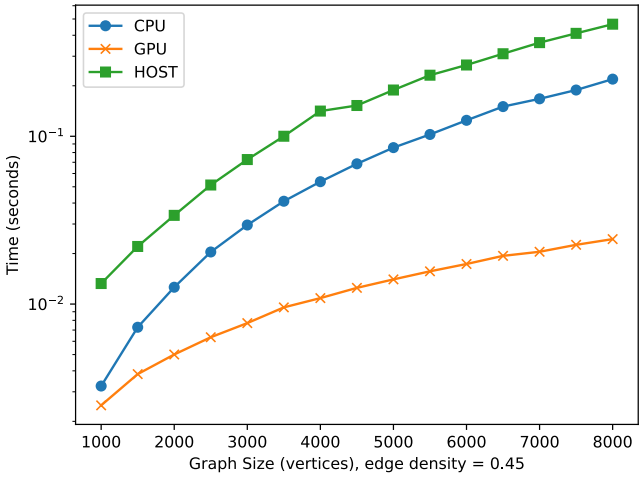
Table for Density 0.65							
	Time (s)			Comparison			
Size	CPU	GPU	HOST	HOST × CPU	GPU × CPU	GPU × HOST	TE
1000	3.49e-01	1.42e-03	6.72e-02	5.19e+00	2.45e+02	4.73e+01	0.3243
1500	1.17e+00	1.95e-03	1.18e-01	9.95e+00	6.01e+02	6.04e+01	0.6216
2000	2.93e+00	2.65e-03	2.09e-01	1.40e+01	1.11e+03	7.91e+01	0.8733
2500	5.96e+00	3.40e-03	3.72e-01	1.60e+01	1.75e+03	1.09e+02	1.0005
3000	1.07e+01	4.39e-03	5.57e-01	1.93e+01	2.45e+03	1.27e+02	1.2056
3500	1.76e+01	5.91e-03	8.07e-01	2.18e+01	2.97e+03	1.37e+02	1.3608
4000	2.59e+01	7.62e-03	1.31e+00	1.98e+01	3.40e+03	1.72e+02	1.2399
4500	3.61e+01	9.16e-03	1.76e+00	2.06e+01	3.94e+03	1.92e+02	1.2846
5000	4.96e+01	1.13e-02	2.07e+00	2.39e+01	4.37e+03	1.83e+02	1.4960
5500	6.57e+01	1.44e-02	3.02e+00	2.17e+01	4.56e+03	2.10e+02	1.3588
6000	8.48e+01	1.74e-02	3.83e+00	2.21e+01	4.87e+03	2.20e+02	1.3828
6500	1.08e+02	2.10e-02	4.79e+00	2.25e+01	5.13e+03	2.28e+02	1.4044
7000	1.36e+02	2.45e-02	5.35e+00	2.54e+01	5.53e+03	2.18e+02	1.5848
7500	1.73e+02	2.90e-02	6.52e+00	2.65e+01	5.97e+03	2.25e+02	1.6588
8000	2.03e+02	3.39e-02	7.82e+00	2.60e+01	6.00e+03	2.31e+02	1.6261

7.5.2 SCC

For strongly connected components, we benchmarked our implementation shown in section 3.5. From the results in figures 7.5 and 7.6, we can conclude that CPU outperforms GPU for small sparse graphs, but GPU soon overtakes it. This seems logical, as the 2 core parts of the algorithm - **BFS** and **comparing** of predecessors and descendants (see section 3.4) - should benefit from parallel execution, although this is not reflected in the HOST results.

Figure 7.5: Comparison of SCC algorithm performance on CPU × GPU on different edge densities





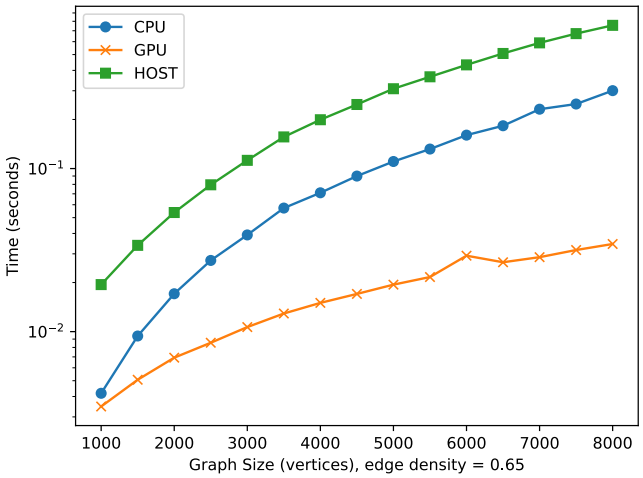


Figure 7.6: Tables for the SCC algorithm graphs in figure 7.5

Table for Density 0.1							
	Time (s)			Comparison			
Size	CPU	GPU	HOST	HOST × CPU	GPU × CPU	GPU × HOST	TE
1000	4.66e-04	9.89e-04	8.00e-03	5.83e-02	4.72e-01	8.10e+00	0.0036
1500	1.03e-03	1.21e-03	8.69e-03	1.19e-01	8.52e-01	7.16e+00	0.0074
2000	1.79e-03	1.74e-03	1.07e-02	1.68e-01	1.03e+00	6.14e+00	0.0105
2500	2.83e-03	2.11e-03	1.22e-02	2.32e-01	1.34e+00	5.78e+00	0.0145
3000	3.93e-03	2.45e-03	1.75e-02	2.25e-01	1.61e+00	7.13e+00	0.0141
3500	5.42e-03	3.27e-03	1.95e-02	2.78e-01	1.66e+00	5.96e+00	0.0174
4000	7.01e-03	3.32e-03	2.49e-02	2.81e-01	2.12e+00	7.52e+00	0.0176
4500	8.98e-03	3.71e-03	2.99e-02	3.00e-01	2.42e+00	8.08e+00	0.0187
5000	1.15e-02	4.00e-03	3.78e-02	3.05e-01	2.89e+00	9.46e+00	0.0191
5500	1.43e-02	4.64e-03	4.49e-02	3.18e-01	3.08e+00	9.67e+00	0.0199
6000	1.72e-02	4.82e-03	5.13e-02	3.36e-01	3.58e+00	1.07e+01	0.0210
6500	2.05e-02	5.27e-03	6.06e-02	3.39e-01	3.90e+00	1.15e+01	0.0212
7000	2.40e-02	5.58e-03	6.81e-02	3.53e-01	4.30e+00	1.22e+01	0.0221
7500	2.80e-02	5.87e-03	7.58e-02	3.69e-01	4.77e+00	1.29e+01	0.0231
8000	3.25e-02	6.27e-03	8.70e-02	3.73e-01	5.19e+00	1.39e+01	0.0233

Table for Density 0.45							
	Time (s)			Comparison			
Size	CPU	GPU	HOST	HOST × CPU	GPU × CPU	GPU × HOST	TE
1000	3.25e-03	2.49e-03	1.33e-02	2.45e-01	1.30e+00	5.32e+00	0.0153
1500	7.27e-03	3.82e-03	2.20e-02	3.30e-01	1.90e+00	5.76e+00	0.0206
2000	1.26e-02	5.01e-03	3.38e-02	3.73e-01	2.52e+00	6.75e+00	0.0233
2500	2.04e-02	6.34e-03	5.12e-02	3.99e-01	3.22e+00	8.08e+00	0.0249
3000	2.96e-02	7.70e-03	7.27e-02	4.07e-01	3.84e+00	9.43e+00	0.0255
3500	4.10e-02	9.56e-03	1.00e-01	4.10e-01	4.29e+00	1.05e+01	0.0256
4000	5.36e-02	1.09e-02	1.41e-01	3.79e-01	4.94e+00	1.30e+01	0.0237
4500	6.85e-02	1.25e-02	1.53e-01	4.49e-01	5.49e+00	1.22e+01	0.0281
5000	8.56e-02	1.40e-02	1.89e-01	4.54e-01	6.10e+00	1.34e+01	0.0283
5500	1.03e-01	1.57e-02	2.31e-01	4.43e-01	6.54e+00	1.48e+01	0.0277
6000	1.24e-01	1.73e-02	2.66e-01	4.68e-01	7.18e+00	1.53e+01	0.0293
6500	1.50e-01	1.94e-02	3.10e-01	4.84e-01	7.75e+00	1.60e+01	0.0303
7000	1.67e-01	2.05e-02	3.62e-01	4.63e-01	8.16e+00	1.76e+01	0.0289
7500	1.89e-01	2.25e-02	4.11e-01	4.59e-01	8.37e+00	1.82e+01	0.0287
8000	2.19e-01	2.44e-02	4.66e-01	4.71e-01	8.98e+00	1.91e+01	0.0294

Table for Density 0.65							
	Time (s)			Comparison			
Size	CPU	GPU	HOST	HOST × CPU	GPU × CPU	GPU × HOST	TE
1000	4.19e-03	3.47e-03	1.94e-02	2.15e-01	1.20e+00	5.59e+00	0.0135
1500	9.40e-03	5.07e-03	3.38e-02	2.78e-01	1.85e+00	6.67e+00	0.0174
2000	1.71e-02	6.92e-03	5.38e-02	3.18e-01	2.47e+00	7.76e+00	0.0199
2500	2.73e-02	8.55e-03	7.95e-02	3.44e-01	3.20e+00	9.30e+00	0.0215
3000	3.92e-02	1.07e-02	1.12e-01	3.49e-01	3.67e+00	1.05e+01	0.0218
3500	5.72e-02	1.29e-02	1.56e-01	3.66e-01	4.43e+00	1.21e+01	0.0229
4000	7.10e-02	1.50e-02	1.99e-01	3.56e-01	4.73e+00	1.33e+01	0.0223
4500	8.99e-02	1.70e-02	2.47e-01	3.64e-01	5.28e+00	1.45e+01	0.0227
5000	1.10e-01	1.94e-02	3.09e-01	3.58e-01	5.69e+00	1.59e+01	0.0224
5500	1.32e-01	2.16e-02	3.65e-01	3.60e-01	6.09e+00	1.69e+01	0.0225
6000	1.60e-01	2.92e-02	4.32e-01	3.71e-01	5.49e+00	1.48e+01	0.0232
6500	1.83e-01	2.66e-02	5.08e-01	3.60e-01	6.87e+00	1.91e+01	0.0225
7000	2.31e-01	2.86e-02	5.90e-01	3.92e-01	8.08e+00	2.06e+01	0.0245
7500	2.49e-01	3.17e-02	6.72e-01	3.70e-01	7.86e+00	2.12e+01	0.0231
8000	3.01e-01	3.45e-02	7.56e-01	3.98e-01	8.72e+00	2.19e+01	0.0249

7.5.3 MIS

For maximal independent set, we have focused on 3 implementations (for comparison between them, see section 7.6.2):

- `MIS_base` - Initial implementation, which does not utilize parallel loops over edges, but rather over vertices. It should work with all storage format kernels (not only CSR related ones).
- `MIS_lex3` - An alternative approach to the problem, where we assume that you just want a single maximal independent set of a given graph, and do not care about much more than that. This implementation is deterministic and always finds the same maximal independent set on a given graph (called *lexicographically first* MIS). This inherently makes it very likely to be the fastest, as there is no random selection of vertices, just finding the maximal independent set.
- `MIS_opt3` - Implementation shown in section 4.5.

Regarding `MIS_base` implementation, we would first like to note that we had to add a fail-safe of sorts to prevent large inconsistent spikes in runtime for both CPU executions. Because of the somewhat random selection process employed in the algorithm (see section 4.4), we can sometimes end up with a collection of unpicked vertices with high degrees (ie. very low chance of being selected). In our base implementation, this can lead to a long stall, where no vertices are selected over many iterations. To prevent this, we have created a counter which counts these *failed* selections, and if there is enough of them one after another, all remaining vertices are selected. This does not make the implementation deterministic, however, as triggering this fail-safe is almost sure to happen only after a substantial amount of vertices were already processed. While we believe this issue could happen on any device, it has never once been the case while performing the algorithm on the GPU.

Coming back to the performance of `MIS_base`, we can see in figures 7.7 and 7.8 that GPU manages to outperform other executions by up to 2 orders of magnitude. For more dense graphs, this performance gap decreases slightly, but it is still more than an order of magnitude in size at all times.

Moving on to `MIS_lex3` (see figures 7.9 and 7.10), GPU once again manages to stay ahead almost all the time. Only for the smallest sparse graphs are the CPU executions somewhat comparable. For the rest, though, GPU is an order of magnitude faster.

When looking at `MIS_opt3` (see figures 7.11 and 7.12), we see behavior similar to what we talked about in regards to `MIS_base`. Here, however, stall should not occur, as we are changing our approach to selecting vertices based on how many viable vertices remain. This perhaps points to an incorrect configuration of OpenMP, or some other issue related to it, as CPU seems to be more or less stable. GPU once again outperforms both CPU and HOST and is also the most stable one.

Figure 7.7: Comparison of `MIS_base` algorithm performance on all devices on different edge densities

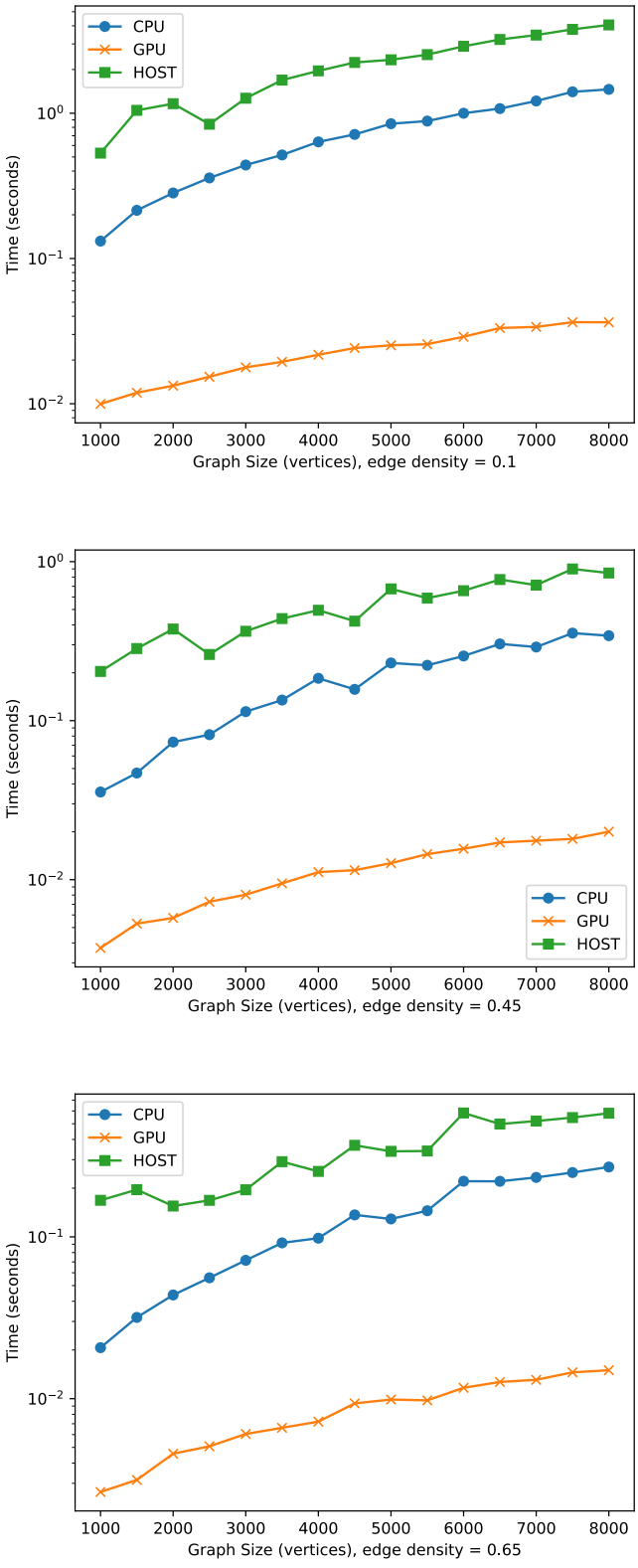


Figure 7.8: Tables for `MIS_base` graphs in figure 7.7

Table for Density 0.1							
	Time (s)			Comparison			
Size	CPU	GPU	HOST	HOST × CPU	GPU × CPU	GPU × HOST	TE
1000	1.32e-01	9.97e-03	5.31e-01	2.48e-01	1.32e+01	5.33e+01	0.0155
1500	2.15e-01	1.19e-02	1.05e+00	2.05e-01	1.80e+01	8.79e+01	0.0128
2000	2.83e-01	1.33e-02	1.16e+00	2.43e-01	2.12e+01	8.74e+01	0.0152
2500	3.59e-01	1.53e-02	8.40e-01	4.27e-01	2.34e+01	5.48e+01	0.0267
3000	4.40e-01	1.78e-02	1.27e+00	3.47e-01	2.47e+01	7.13e+01	0.0217
3500	5.16e-01	1.94e-02	1.69e+00	3.05e-01	2.66e+01	8.71e+01	0.0191
4000	6.35e-01	2.17e-02	1.96e+00	3.24e-01	2.92e+01	9.01e+01	0.0203
4500	7.15e-01	2.42e-02	2.24e+00	3.20e-01	2.95e+01	9.25e+01	0.0200
5000	8.47e-01	2.53e-02	2.33e+00	3.63e-01	3.35e+01	9.24e+01	0.0227
5500	8.83e-01	2.57e-02	2.53e+00	3.49e-01	3.44e+01	9.85e+01	0.0218
6000	1.00e+00	2.89e-02	2.89e+00	3.46e-01	3.46e+01	1.00e+02	0.0216
6500	1.08e+00	3.32e-02	3.22e+00	3.34e-01	3.24e+01	9.69e+01	0.0209
7000	1.21e+00	3.38e-02	3.45e+00	3.52e-01	3.59e+01	1.02e+02	0.0220
7500	1.40e+00	3.64e-02	3.78e+00	3.72e-01	3.86e+01	1.04e+02	0.0232
8000	1.46e+00	3.64e-02	4.06e+00	3.60e-01	4.01e+01	1.11e+02	0.0225

Table for Density 0.45							
	Time (s)			Comparison			
Size	CPU	GPU	HOST	HOST × CPU	GPU × CPU	GPU × HOST	TE
1000	3.56e-02	3.72e-03	2.04e-01	1.75e-01	9.57e+00	5.48e+01	0.0109
1500	4.69e-02	5.29e-03	2.84e-01	1.65e-01	8.86e+00	5.37e+01	0.0103
2000	7.34e-02	5.74e-03	3.77e-01	1.95e-01	1.28e+01	6.56e+01	0.0122
2500	8.16e-02	7.27e-03	2.61e-01	3.13e-01	1.12e+01	3.59e+01	0.0196
3000	1.14e-01	8.04e-03	3.65e-01	3.12e-01	1.42e+01	4.54e+01	0.0195
3500	1.35e-01	9.48e-03	4.38e-01	3.07e-01	1.42e+01	4.63e+01	0.0192
4000	1.85e-01	1.12e-02	4.95e-01	3.73e-01	1.66e+01	4.44e+01	0.0233
4500	1.57e-01	1.15e-02	4.23e-01	3.72e-01	1.37e+01	3.68e+01	0.0233
5000	2.31e-01	1.27e-02	6.74e-01	3.42e-01	1.82e+01	5.31e+01	0.0214
5500	2.23e-01	1.45e-02	5.90e-01	3.78e-01	1.54e+01	4.08e+01	0.0236
6000	2.50e-01	1.56e-02	6.57e-01	3.82e-01	1.60e+01	4.19e+01	0.0238
6500	3.04e-01	1.71e-02	7.72e-01	3.94e-01	1.77e+01	4.50e+01	0.0246
7000	2.90e-01	1.76e-02	7.12e-01	4.08e-01	1.65e+01	4.05e+01	0.0255
7500	3.55e-01	1.80e-02	9.00e-01	3.95e-01	1.97e+01	4.98e+01	0.0247
8000	3.42e-01	2.00e-02	8.48e-01	4.03e-01	1.71e+01	4.23e+01	0.0252

Table for Density 0.65							
	Time (s)			Comparison			
Size	CPU	GPU	HOST	HOST × CPU	GPU × CPU	GPU × HOST	TE
1000	2.07e-02	2.65e-03	1.68e-01	1.23e-01	7.79e+00	6.34e+01	0.0077
1500	3.18e-02	3.15e-03	1.96e-01	1.62e-01	1.01e+01	6.22e+01	0.0102
2000	4.37e-02	4.57e-03	1.55e-01	2.82e-01	9.57e+00	3.39e+01	0.0176
2500	5.59e-02	5.08e-03	1.68e-01	3.32e-01	1.10e+01	3.31e+01	0.0208
3000	7.16e-02	6.05e-03	1.95e-01	3.67e-01	1.18e+01	3.23e+01	0.0229
3500	9.18e-02	6.61e-03	2.91e-01	3.15e-01	1.39e+01	4.41e+01	0.0197
4000	9.81e-02	7.21e-03	2.54e-01	3.87e-01	1.36e+01	3.52e+01	0.0242
4500	1.37e-01	9.35e-03	3.68e-01	3.72e-01	1.46e+01	3.94e+01	0.0232
5000	1.29e-01	9.88e-03	3.38e-01	3.82e-01	1.31e+01	3.42e+01	0.0239
5500	1.45e-01	9.76e-03	3.39e-01	4.27e-01	1.48e+01	3.47e+01	0.0267
6000	2.21e-01	1.17e-02	5.83e-01	3.79e-01	1.89e+01	4.99e+01	0.0237
6500	2.20e-01	1.27e-02	4.98e-01	4.43e-01	1.74e+01	3.93e+01	0.0277
7000	2.33e-01	1.31e-02	5.20e-01	4.48e-01	1.78e+01	3.97e+01	0.0280
7500	2.50e-01	1.45e-02	5.46e-01	4.57e-01	1.72e+01	3.75e+01	0.0286
8000	2.70e-01	1.50e-02	5.81e-01	4.65e-01	1.80e+01	3.87e+01	0.0291

Figure 7.9: Comparison of `MIS_lex3` algorithm performance on all devices on different edge densities.

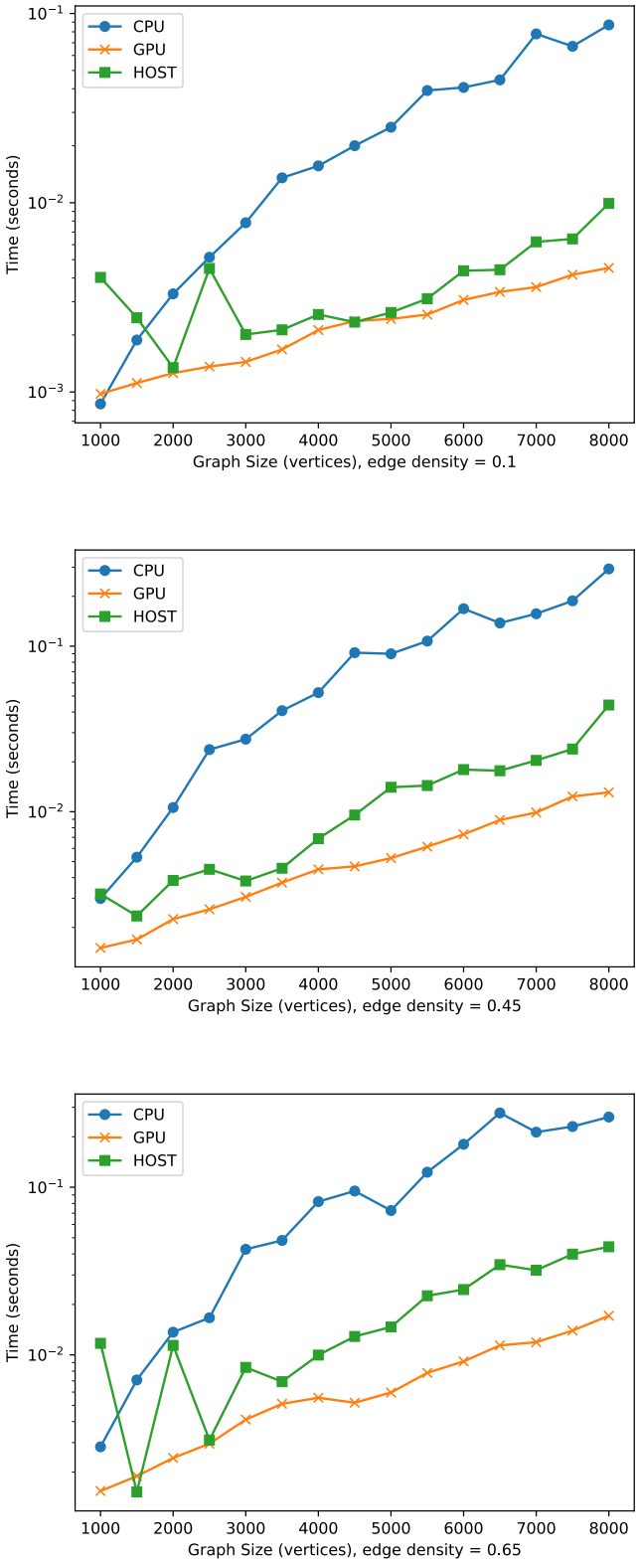


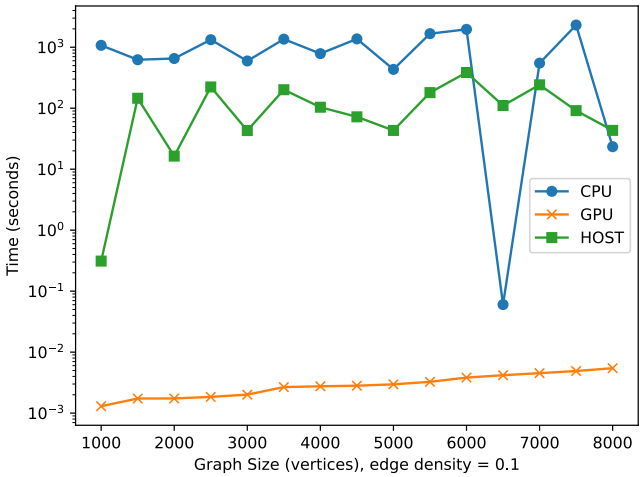
Figure 7.10: Tables for `MIS_lex3` graphs in figure 7.9

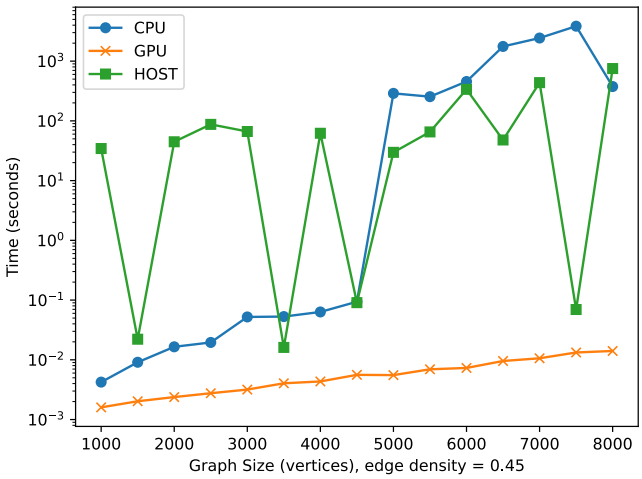
Table for Density 0.1							
	Time (s)			Comparison			
Size	CPU	GPU	HOST	HOST × CPU	GPU × CPU	GPU × HOST	TE
1000	8.65e-04	9.77e-04	4.03e-03	2.15e-01	8.86e-01	4.13e+00	0.0134
1500	1.88e-03	1.11e-03	2.47e-03	7.62e-01	1.69e+00	2.21e+00	0.0476
2000	3.30e-03	1.26e-03	1.35e-03	2.45e+00	2.62e+00	1.07e+00	0.1534
2500	5.15e-03	1.36e-03	4.49e-03	1.15e+00	3.78e+00	3.30e+00	0.0716
3000	7.84e-03	1.44e-03	2.01e-03	3.89e+00	5.44e+00	1.40e+00	0.2433
3500	1.35e-02	1.68e-03	2.13e-03	6.36e+00	8.07e+00	1.27e+00	0.3974
4000	1.57e-02	2.12e-03	2.57e-03	6.10e+00	7.37e+00	1.21e+00	0.3812
4500	2.00e-02	2.37e-03	2.34e-03	8.56e+00	8.45e+00	9.87e-01	0.5348
5000	2.51e-02	2.43e-03	2.63e-03	9.53e+00	1.03e+01	1.08e+00	0.5957
5500	3.92e-02	2.56e-03	3.11e-03	1.26e+01	1.53e+01	1.21e+00	0.7872
6000	4.07e-02	3.06e-03	4.37e-03	9.31e+00	1.33e+01	1.43e+00	0.5817
6500	4.46e-02	3.38e-03	4.42e-03	1.01e+01	1.32e+01	1.31e+00	0.6297
7000	7.80e-02	3.58e-03	6.21e-03	1.26e+01	2.18e+01	1.73e+00	0.7851
7500	6.70e-02	4.16e-03	6.44e-03	1.04e+01	1.61e+01	1.55e+00	0.6509
8000	8.70e-02	4.52e-03	9.91e-03	8.78e+00	1.92e+01	2.19e+00	0.5489

Table for Density 0.45							
	Time (s)			Comparison			
Size	CPU	GPU	HOST	HOST × CPU	GPU × CPU	GPU × HOST	TE
1000	2.99e-03	1.50e-03	3.18e-03	9.39e-01	1.99e+00	2.12e+00	0.0587
1500	5.31e-03	1.69e-03	2.34e-03	2.26e+00	3.14e+00	1.39e+00	0.1415
2000	1.06e-02	2.24e-03	3.84e-03	2.76e+00	4.72e+00	1.71e+00	0.1724
2500	2.37e-02	2.57e-03	4.49e-03	5.29e+00	9.23e+00	1.74e+00	0.3305
3000	2.74e-02	3.06e-03	3.81e-03	7.19e+00	8.97e+00	1.25e+00	0.4495
3500	4.08e-02	3.73e-03	4.55e-03	8.96e+00	1.09e+01	1.22e+00	0.5597
4000	5.24e-02	4.48e-03	6.87e-03	7.62e+00	1.17e+01	1.53e+00	0.4764
4500	9.14e-02	4.67e-03	9.55e-03	9.57e+00	1.95e+01	2.04e+00	0.5981
5000	8.99e-02	5.24e-03	1.40e-02	6.40e+00	1.72e+01	2.68e+00	0.4000
5500	1.07e-01	6.14e-03	1.44e-02	7.46e+00	1.75e+01	2.34e+00	0.4662
6000	1.66e-01	7.29e-03	1.80e-02	9.26e+00	2.28e+01	2.46e+00	0.5786
6500	1.38e-01	8.91e-03	1.77e-02	7.81e+00	1.55e+01	1.98e+00	0.4881
7000	1.57e-01	9.88e-03	2.04e-02	7.69e+00	1.59e+01	2.07e+00	0.4805
7500	1.88e-01	1.24e-02	2.39e-02	7.85e+00	1.52e+01	1.94e+00	0.4908
8000	2.93e-01	1.31e-02	4.41e-02	6.64e+00	2.24e+01	3.37e+00	0.4151

Table for Density 0.65							
	Time (s)			Comparison			
Size	CPU	GPU	HOST	HOST × CPU	GPU × CPU	GPU × HOST	TE
1000	2.83e-03	1.54e-03	1.17e-02	2.41e-01	1.83e+00	7.60e+00	0.0151
1500	7.09e-03	1.89e-03	1.52e-03	4.66e+00	3.75e+00	8.05e-01	0.2911
2000	1.37e-02	2.43e-03	1.14e-02	1.20e+00	5.63e+00	4.69e+00	0.0750
2500	1.67e-02	2.95e-03	3.10e-03	5.36e+00	5.64e+00	1.05e+00	0.3353
3000	4.27e-02	4.11e-03	8.42e-03	5.07e+00	1.04e+01	2.05e+00	0.3166
3500	4.82e-02	5.11e-03	6.92e-03	6.96e+00	9.43e+00	1.36e+00	0.4348
4000	8.21e-02	5.55e-03	9.98e-03	8.23e+00	1.48e+01	1.80e+00	0.5143
4500	9.50e-02	5.18e-03	1.29e-02	7.37e+00	1.83e+01	2.48e+00	0.4609
5000	7.26e-02	5.96e-03	1.47e-02	4.95e+00	1.22e+01	2.46e+00	0.3092
5500	1.23e-01	7.81e-03	2.25e-02	5.46e+00	1.57e+01	2.88e+00	0.3412
6000	1.80e-01	9.14e-03	2.45e-02	7.36e+00	1.97e+01	2.68e+00	0.4597
6500	2.77e-01	1.14e-02	3.45e-02	8.03e+00	2.44e+01	3.03e+00	0.5021
7000	2.13e-01	1.19e-02	3.20e-02	6.66e+00	1.79e+01	2.69e+00	0.4164
7500	2.30e-01	1.40e-02	3.99e-02	5.77e+00	1.65e+01	2.86e+00	0.3604
8000	2.62e-01	1.71e-02	4.41e-02	5.94e+00	1.53e+01	2.58e+00	0.3715

Figure 7.11: Comparison of `MIS_opt3` algorithm performance on all devices on different edge densities





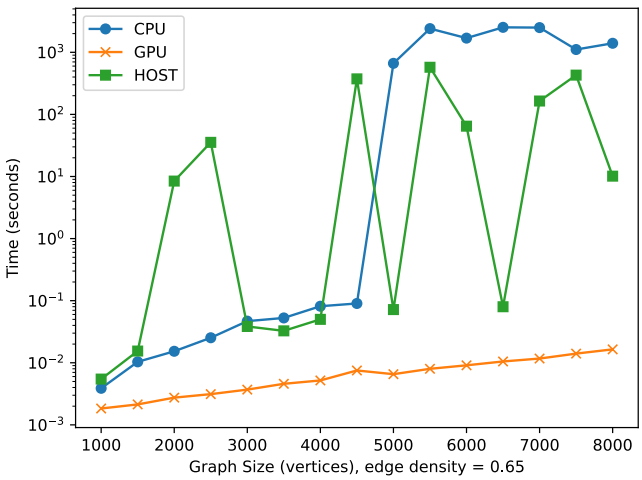


Figure 7.12: Tables for `MIS_opt3` graphs in figure 7.11

Table for Density 0.1							
	Time (s)			Comparison			
Size	CPU	GPU	HOST	HOST × CPU	GPU × CPU	GPU × HOST	TE
1000	1.07e+03	1.30e-03	3.11e-01	3.45e+03	8.27e+05	2.40e+02	215.7109
1500	6.24e+02	1.74e-03	1.46e+02	4.28e+00	3.59e+05	8.38e+04	0.2678
2000	6.53e+02	1.74e-03	1.64e+01	3.98e+01	3.75e+05	9.43e+03	2.4886
2500	1.33e+03	1.85e-03	2.23e+02	5.95e+00	7.20e+05	1.21e+05	0.3719
3000	5.95e+02	2.01e-03	4.30e+01	1.38e+01	2.96e+05	2.14e+04	0.8651
3500	1.36e+03	2.67e-03	2.01e+02	6.75e+00	5.08e+05	7.53e+04	0.4218
4000	7.85e+02	2.75e-03	1.04e+02	7.56e+00	2.85e+05	3.77e+04	0.4728
4500	1.37e+03	2.82e-03	7.22e+01	1.90e+01	4.86e+05	2.56e+04	1.1848
5000	4.33e+02	2.96e-03	4.32e+01	1.00e+01	1.46e+05	1.46e+04	0.6270
5500	1.67e+03	3.26e-03	1.80e+02	9.26e+00	5.12e+05	5.53e+04	0.5787
6000	1.96e+03	3.83e-03	3.85e+02	5.10e+00	5.13e+05	1.01e+05	0.3186
6500	6.03e-02	4.18e-03	1.11e+02	5.45e-04	1.44e+01	2.64e+04	0.0000
7000	5.48e+02	4.52e-03	2.42e+02	2.27e+00	1.21e+05	5.34e+04	0.1417
7500	2.31e+03	4.90e-03	9.12e+01	2.53e+01	4.71e+05	1.86e+04	1.5829
8000	2.34e+01	5.45e-03	4.33e+01	5.40e-01	4.29e+03	7.94e+03	0.0337

Table for Density 0.45							
	Time (s)			Comparison			
Size	CPU	GPU	HOST	HOST × CPU	GPU × CPU	GPU × HOST	TE
1000	4.23e-03	1.60e-03	3.44e+01	1.23e-04	2.65e+00	2.15e+04	0.0000
1500	9.10e-03	2.02e-03	2.21e-02	4.11e-01	4.49e+00	1.09e+01	0.0257
2000	1.65e-02	2.38e-03	4.47e+01	3.69e-04	6.94e+00	1.88e+04	0.0000
2500	1.95e-02	2.76e-03	8.74e+01	2.23e-04	7.08e+00	3.17e+04	0.0000
3000	5.22e-02	3.18e-03	6.64e+01	7.86e-04	1.64e+01	2.09e+04	0.0000
3500	5.30e-02	4.05e-03	1.61e-02	3.28e+00	1.31e+01	3.99e+00	0.2051
4000	6.33e-02	4.34e-03	6.20e+01	1.02e-03	1.46e+01	1.43e+04	0.0001
4500	9.38e-02	5.60e-03	9.05e-02	1.04e+00	1.67e+01	1.62e+01	0.0648
5000	2.89e+02	5.54e-03	2.97e+01	9.71e+00	5.21e+04	5.36e+03	0.6069
5500	2.54e+02	6.94e-03	6.53e+01	3.89e+00	3.66e+04	9.41e+03	0.2432
6000	4.56e+02	7.31e-03	3.37e+02	1.35e+00	6.24e+04	4.62e+04	0.0845
6500	1.76e+03	9.56e-03	4.79e+01	3.68e+01	1.84e+05	5.01e+03	2.2995
7000	2.44e+03	1.06e-02	4.37e+02	5.58e+00	2.30e+05	4.11e+04	0.3488
7500	3.85e+03	1.33e-02	6.95e-02	5.53e+04	2.90e+05	5.24e+00	3458.5100
8000	3.75e+02	1.41e-02	7.52e+02	4.99e-01	2.67e+04	5.34e+04	0.0312

Table for Density 0.65							
	Time (s)			Comparison			
Size	CPU	GPU	HOST	HOST × CPU	GPU × CPU	GPU × HOST	TE
1000	3.88e-03	1.83e-03	5.46e-03	7.10e-01	2.11e+00	2.98e+00	0.0444
1500	1.03e-02	2.14e-03	1.54e-02	6.70e-01	4.84e+00	7.23e+00	0.0419
2000	1.53e-02	2.74e-03	8.44e+00	1.82e-03	5.58e+00	3.07e+03	0.0001
2500	2.53e-02	3.13e-03	3.55e+01	7.13e-04	8.09e+00	1.13e+04	0.0000
3000	4.68e-02	3.70e-03	3.84e-02	1.22e+00	1.27e+01	1.04e+01	0.0762
3500	5.27e-02	4.59e-03	3.27e-02	1.61e+00	1.15e+01	7.12e+00	0.1009
4000	8.14e-02	5.19e-03	5.00e-02	1.63e+00	1.57e+01	9.64e+00	0.1017
4500	9.02e-02	7.52e-03	3.73e+02	2.41e-04	1.20e+01	4.97e+04	0.0000
5000	6.64e+02	6.55e-03	7.19e-02	9.24e+03	1.01e+05	1.10e+01	577.5891
5500	2.41e+03	8.00e-03	5.74e+02	4.19e+00	3.01e+05	7.18e+04	0.2620
6000	1.69e+03	9.10e-03	6.46e+01	2.62e+01	1.86e+05	7.10e+03	1.6363
6500	2.52e+03	1.05e-02	7.99e-02	3.16e+04	2.40e+05	7.61e+00	1975.3964
7000	2.50e+03	1.17e-02	1.64e+02	1.52e+01	2.14e+05	1.40e+04	0.9513
7500	1.11e+03	1.40e-02	4.29e+02	2.58e+00	7.89e+04	3.06e+04	0.1614
8000	1.40e+03	1.64e-02	1.01e+01	1.38e+02	8.50e+04	6.16e+02	8.6230

7.6 Optimizations

7.6.1 CC

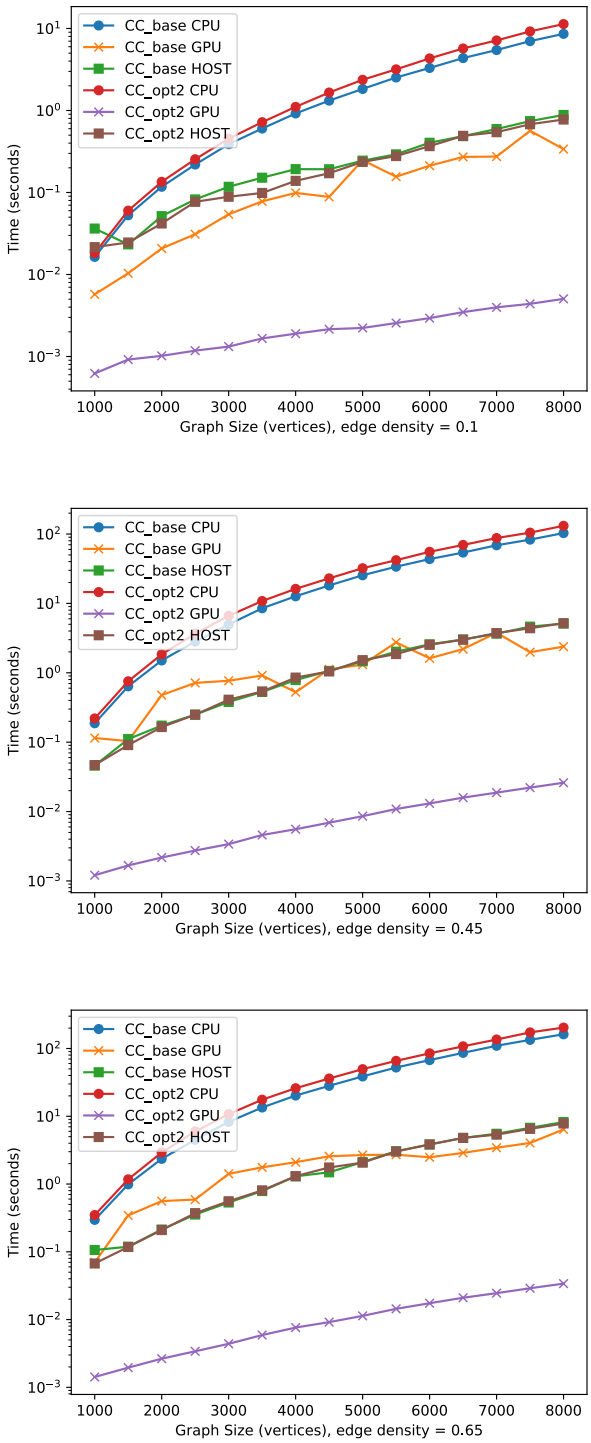
For connected components, we are going to compare 2 implementations of our algorithm:

- `CC_base` - Initial implementation, which does not utilize parallel loops over edges, but rather over vertices. This implementation is not shown here, as we believe it is inferior to the optimized one. Note, however, that it should work with all storage format kernels. For exact times, see table 7.2.
- `CC_opt2` - Implementation shown in section 2.5. For exact times, see table 7.4.

When we look at how they perform side-by-side, we discover that each device is affected differently by the changes made between implementations (see figure 7.13).

While `CC_opt2` represents a major improvement for the GPU, it actually hinders sequential execution performance sightly. HOST seems to not be affected much. Considering the huge performance improvement for the GPU, which is the main focus of our work, we believe `CC_opt2` to be the better implementation of the 2.

Figure 7.13: Comparing `CC_base` graphs with `CC_opt2` graphs.



7.6.2 MIS

For maximal independent set, we are going to compare 3 implementations of our algorithm:

- `MIS_base` - Initial implementation, which does not utilize parallel loops over edges, but rather over vertices. It should work with all storage format kernels (not only CSR related ones).
- `MIS_lex3` - An alternative approach to the problem, where we assume that you just want a single maximal independent set of a given graph, and do not care about much more than that. This implementation is deterministic and always finds the same maximal independent set on a given graph (called *lexicographically first* MIS). This inherently makes it very likely to be the fastest, as there is no random selection of vertices, just finding the maximal independent set.
- `MIS_opt3` - Implementation shown in section 4.5.

For readability and convenience, let us split comparisons into pairs.

When comparing `MIS_base` with `MIS_opt3` (see figure 7.14), we can see that `MIS_opt3` improves the performance on the GPU by roughly an order of magnitude for less dense graphs. As edge density increases, both implementations become more and more comparable to each other. A huge difference between the two, however, can be seen on CPU. We can observe that `MIS_opt3` seems to exhibit unreliable performance using OpenMP, as the edge density increases. Inconsistencies in performance are somewhat puzzling, as using the adaptive approach to vertex selection (see section 4.5) should in theory prevent this behaviour. This could perhaps hint at some mistake in OpenMP configuration, as sequential benchmark seems to not behave like this. While `MIS_opt3` offers better performance on the GPU, `MIS_base` seems to offer more reliable performance overall. Should someone be certain to use GPU, however, `MIS_opt3` seems to have no drawbacks when running on it.

Next, let us compare `MIS_base` with `MIS_lex3` (see figure 7.15). It can be observed, that `MIS_lex3` is faster than `MIS_base`, which is to be expected. With increasing graph size and edge density, we can see both implementation become more and more comparable in all cases except for HOST, where `MIS_lex3` manages to stay significantly faster than `MIS_base`.

Lastly, when comparing `MIS_opt3` with `MIS_lex3` (see figure 7.16), we can see that `MIS_lex3` is faster and more reliable in terms of performance, especially on both CPU executions. Because of the large y axis range, GPU results are somewhat skewed, so they are shown on their own in figure 7.17. There, it can be seen, that they are quite comparable, especially for dense graphs. So while on both CPU executions, `MIS_lex3` performs much better than `MIS_opt3` (and also quite better than `MIS_base`), GPU performance looks about the same for both. This is interesting, as it implies, that the overhead associated with random selection of vertices is different on each device. It is very small on the GPU, while it is much larger on the CPU based executions.

Overall, `MIS_lex3` seems like the fastest option for both HOST and CPU, but comes with the drawback of always resulting in the same set on a given graph. Should a random set be required, `MIS_base` seems to be the best option for HOST and CPU. For the GPU, `MIS_opt3` performs the best, while offering random result.

Figure 7.14: Comparing `MIS_base` graphs with `MIS_opt3` graphs.

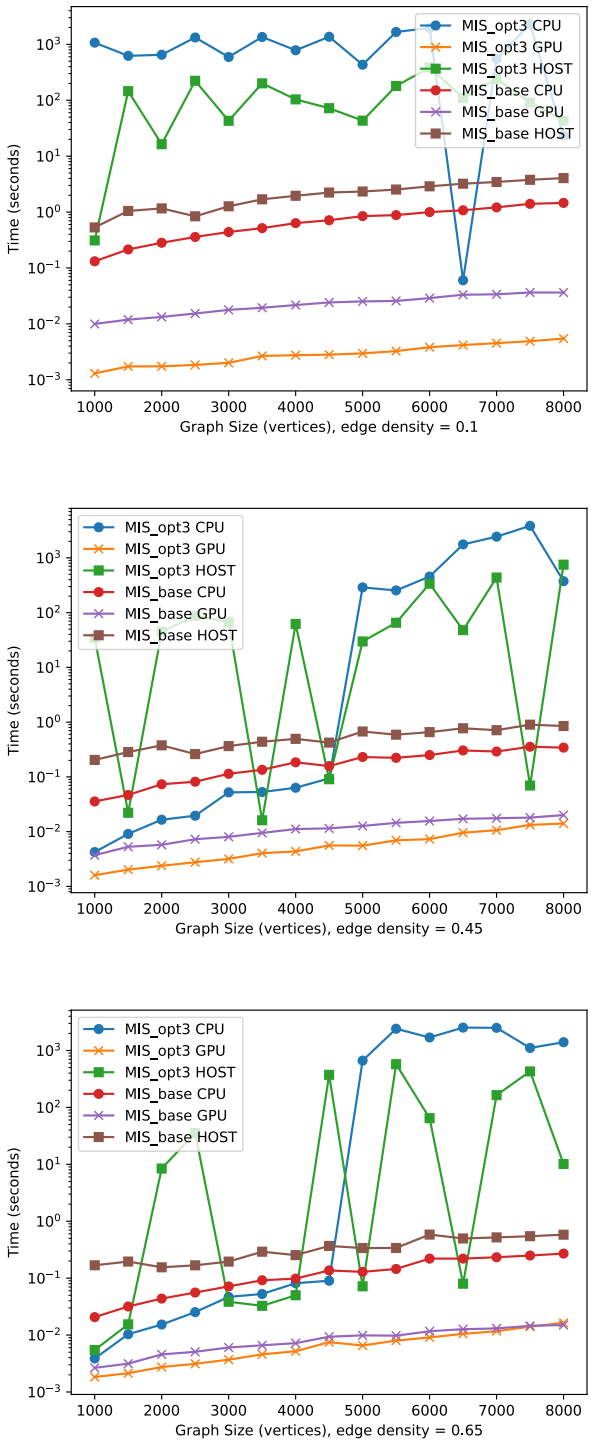


Figure 7.15: Comparing `MIS_base` graphs with `MIS_lex3` graphs.

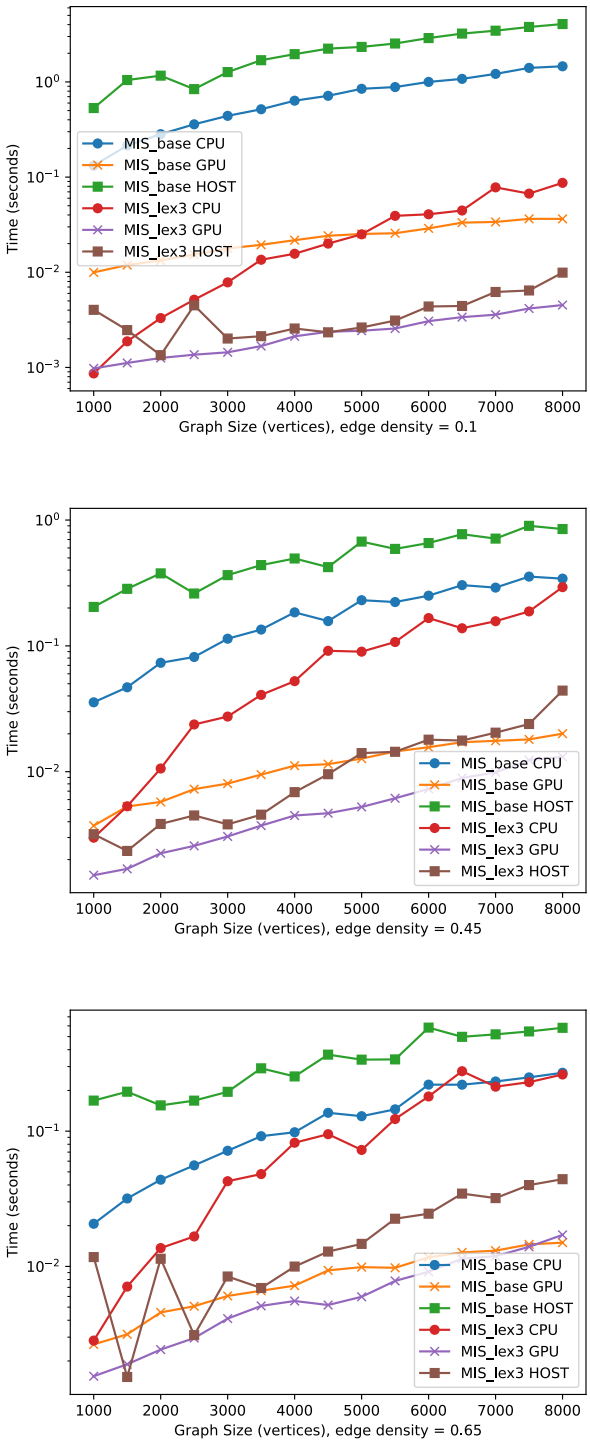


Figure 7.16: Comparing `MIS_opt3` graphs with `MIS_lex3` graphs.

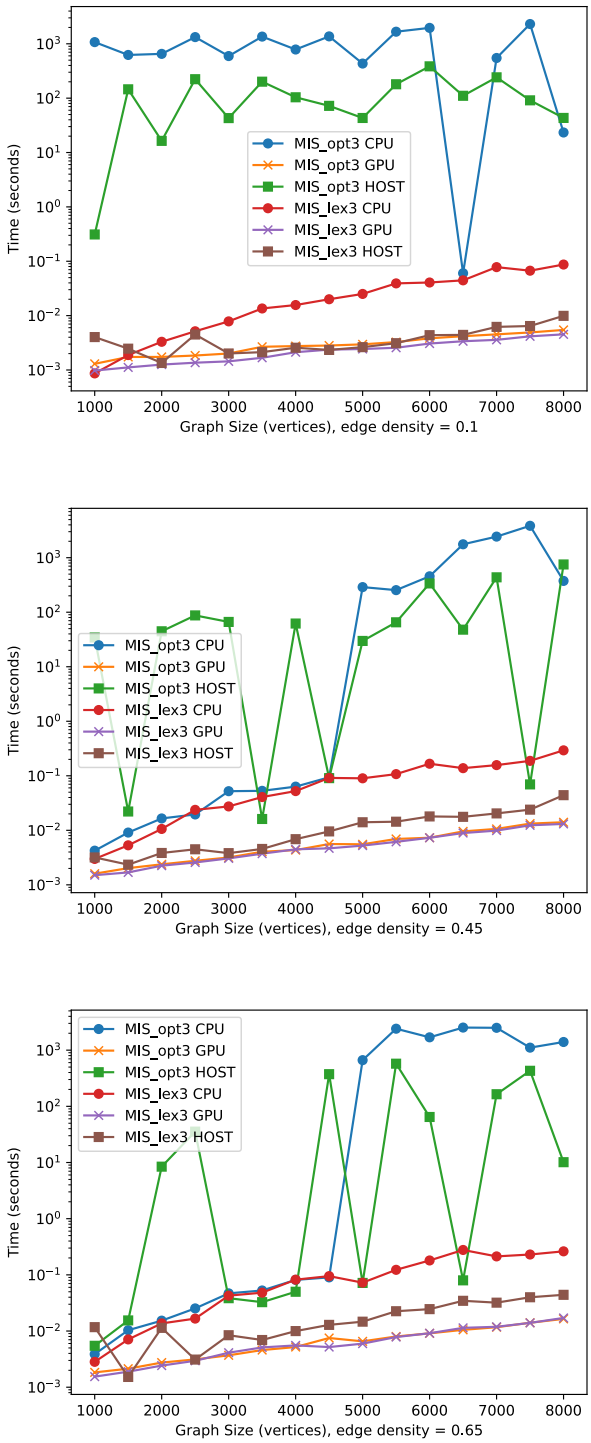
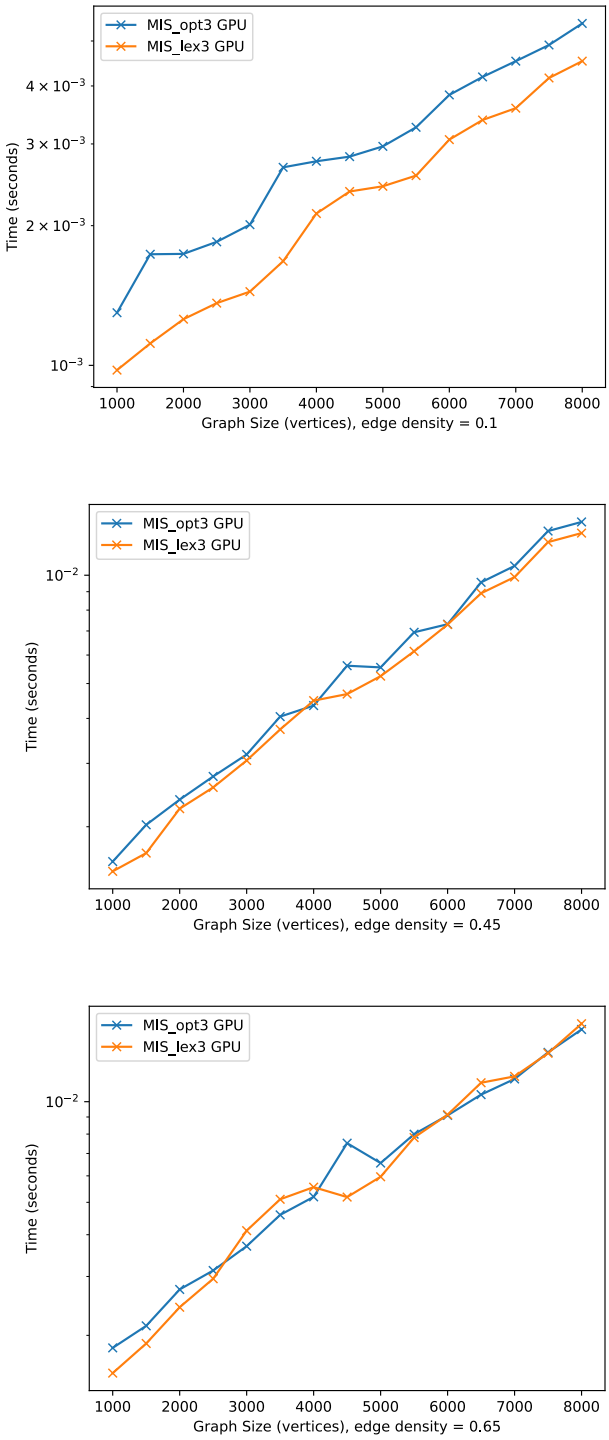


Figure 7.17: Comparing `MIS_opt3` graphs with `MIS_lex3` graphs on GPU.



Conclusion

Goal of this project was to select, describe, test, implement, and benchmark parallel algorithms for solving common graph problems, namely finding maximal independent set, (strongly) connected components, and minimal spanning tree (forest).

To summarize, we have managed to create fully working and well-performing implementations for 3 out of the 4 proposed graph problems, with (hopefully) comprehensive descriptions. For the last problem - minimal spanning tree - we at least offered our *work in progress* implementations, which are not 100% reliable, but they provide some value and can be used. These implementations also add functionality to the TNL library, as they will become part of it. Note that no implementations were present in TNL to solve these problems before. Furthermore, some tools created for the purposes of this work, such as a Python script for generating visualizations of graph benchmark results, can be used for future graph algorithms. We would also like to point out the theoretical parts and algorithm descriptions, which we believe were well written and illustrated, offering understandable and concise summary of these graph problems and solutions. All implementations and code produced in the process of making this thesis can be freely accessed on GitLab [cic24], where it is currently in branch `RC/GraphAlgorithms`, awaiting merging procedure. In the future, it is assumed to be present as official part of TNL.

Still, it is unfortunate, that not every goal was accomplished. To address our shortcomings, we would like to list some parts of our work that we feel perhaps lack the depth and detail they deserve, whether it be due to unexpected problems, not ideal time management, or any other reason:

- **External libraries -** We consider this to probably be the most severe part that is lacking, as having any sort of comparison with another library would elevate the whole benchmarking discussion and provide better, more objective, and more informative comparison, allowing us to better frame our implementation performances. We were simply unable to make the libraries function on our system and within our benchmarking environment in a timely manner, when it came to benchmarking. A clear lesson was taken from this - allocate more time to first confirm the functionality of and prepare the libraries on the system benchmarks are measured on, even if it means that there will be less time to tweak or work on the implementations of various algorithms or other parts of thesis. It would perhaps be better to have a more complete comparison with external libraries, even at the cost of not being able to develop our implementations as much.
- **Sequential algorithm benchmarking -** We could have created and measured standard sequential only implementations to compare with out parallel focused ones to get a clearer comparison between the 2 hardware architectures. To be fair, this was not the main focus of our work, as we were more hoping to explore the parallel implementations and focus on GPU performance. Since we were unable to compare with external libraries, though, the lack of clear CPU to GPU comparison also becomes problematic, as all of our implementations should implicitly favor parallel execution,

because their logic is based around it. Comparing against an implementation made with sequential execution in mind could have provided better device comparison.

- **Measuring with real graphs -** We did not perform the benchmarks on any *organic* graphs from real data, mainly due to time constrains. While we believe measuring on generated graphs offers better conditions for observing performance impact of certain graph characteristics, a valid argument can be made about how the performance measured holds up when using our algorithms on real graphs.

With the ability to now reflect on the process of creating this thesis, we still firmly believe that our dedication to and time spend on it was respectable, but that different decisions would perhaps lead to better and overall more complete results. Mainly, spending less time on the implementation development and optimization, and more time on the measuring part of the work. This way, a more complete and better framed result could be presented, even if its not as developed or optimized. Should we continue to work on the foundation we laid out in this thesis in the future, we would certainly apply the lessons learned while making it, and try to improve the parts of it that were left behind, starting with putting more emphasis on and dedicating more time to the benchmarking part of the work.

Despite not being able to achieve everything we have set out to do, we still consider this work a successful and valuable endeavour, academic contribution, addition to the TNL library, and a personal experience.

Bibliography

[AI24] RAPIDS AI. cuGraph. `https://github.com/rapidsai/cugraph`, 2024.

[AS87] Awerbuch and Shiloach. New connectivity and msf algorithms for shuffle-exchange network and pram. *IEEE Transactions on Computers*, C-36(10):1258–1263, 1987.

[BKS22] Tim Baer, Raghavendra Kanakagiri, and Edgar Solomonik. Parallel minimum spanning forest computation using sparse matrix kernels. *Proceedings of the 2022 SIAM Conference on Parallel Processing for Scientific Computing (PP)*, page 72–83, Jan. 2022.

[Bor26] O. Borůvka. Über ein Minimalproblem. *Práce moravské přírodovědecké společnosti 3*, 37-58 (1926)., 1926.

[CFHP06] Don Coppersmith, Lisa Fleischer, Bruce Hendrickson, and Ali Pinar. A divide-and-conquer algorithm for identifying strongly connected components. Jan. 2006.

[cic24] cichrrad. Fork of tnl project, branch `RC/GraphAlgorithms`. `https://gitlab.com/cichrrad/tnl-cichrrad`, August 2024. Fork of `https://gitlab.com/tnl-project/tnl`.

[Goo24] Google. Google/googletest: GoogleTest - Google Testing and mocking framework. *GitHub*, 2024.

[HT73] John Hopcroft and Robert Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Commun. ACM*, 16(6):372–378, Jun. 1973.

[KOFv22] Jakub Klinkovský, Tomáš Oberhuber, Radek Fučík, and Vítězslav Žabka. Configurable open-source data structure for distributed conforming unstructured homogeneous meshes with gpu support. *ACM Trans. Math. Softw.*, 48(3), sep 2022.

[Kru56] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48, Feb 1956.

[LLRK80] E. L. Lawler, J. K. Lenstra, and A. H. Rinnooy Kan. Generating all maximal independent sets: Np-hardness and polynomial-time algorithms. *SIAM Journal on Computing*, 9(3):558–565, Aug. 1980.

[Lub86] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15(4):1036–1053, 1986.

[Nvi24] Nvidia. *Cuda C++ Programming Guide*, Mar 2024.

[OKF21] Tomáš Oberhuber, Jakub Klinkovský, and Radek Fučík. Tnl: Numerical library for modern parallel architectures. *Acta Polytechnica*, 61(SI):122–134, Feb. 2021.

[Pri57] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, Nov 1957.

[Rei85] John H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, Jun 1985.

[Sha81] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers Mathematics with Applications*, 7(1):67–72, 1981.

[Str24] Pavel Strachota. Helios cluster documentation. `http://helios.fjfi.cvut.cz/`, 2024. Department of Mathematics, Faculty of Nuclear Sciences and Physical Engineering, Czech Technical University in Prague.

[SV82] Yossi Shiloach and Uzi Vishkin. An o(logn) parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, Mar 1982.

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[Tem24] Template Numerical Library. Tnl user guide - general concepts. `https://tnl-project.gitlab.io/tnl/ug_GeneralConcepts.html#ug_general_concepts_views_and_shared_pointers`, 2024. Accessed: 2024-07-29.

[WPD+17] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens. Gunrock: GPU graph analytics. *ACM Transactions on Parallel Computing*, 4(1):3:1–3:49, August 2017.

[ZAB20] Yongzhe Zhang, Ariful Azad, and Aydın Buluç. Parallel algorithms for finding connected components using linear algebra. *Journal of Parallel and Distributed Computing*, 144:14–27, May 2020.

[ZAH20] Yongzhe Zhang, Ariful Azad, and Zhenjiang Hu. Fastsv: A distributed-memory connected component algorithm with fast convergence. *Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing*, page 46–57, Jan. 2020.