**DENSEST SUBGRAPH DISCOVERY ON THE GPU**

by

Hunter G. Gareau

A Thesis

Submitted to the

(insert department name)

College of Science and Mathematics

In partial fulfillment of the requirement

For the degree of

Master of Science in Computer Science

at

Rowan University

(insert defense date)

Thesis Chair: (insert Professor Guo info)

Committee Members:

(insert Yu info)

(insert Rabbitz info)

© 2025 Hunter G. Gareau

**Acknowledgements**

I would like to deeply thank Professor Guimu Guo for his guidance throughout our research. We have worked together for a long time on this project, and I wouldn’t have been able to do any of this without his skills, direction, and help.

(insert thanks for other Committee members)

(insert thanks for other student?)

**Abstract**

Hunter Gareau

DENSEST SUBGRAPH DISCOVERY ON THE GPU

2024 – 2025

Dr. Guimu Guo

Master of Science in Computer Science

(Abstract starts here)

**Table of Contents**

Abstract iv

List of Figures vi

List of Tables vii

Chapter 1: Introduction 1

Chapter 2: Existing Solutions (tbd)

Chapter 3: The Algorithm (tbd)

Chapter 4:

**List of Figures**

Figure Page

Figure 1. (Title here) (Page #)

List of Tables

Table Page

Table 1. (Title here) (Page #)

**Chapter 1**

**Introduction**

When it comes to analyzing large or complex groups of data, it is often useful to examine the connections and relationships shared between its subjects. Graphs can be used to model such relations. Graphs are composed of two components: vertices (which represent individual members of a data set) and edges (which represent the connections between these members). A graph is usually represented as *G = (V, E)*, where *V* is the set of vertices in the graph, and *E* is the set of edges in the graph. A couple of the major types of data that graphs can be used to model are social networks (e.g. Facebook, Twitter, etc.) and biological data (DNA, neural networks, etc.) [2]. It’s also worth noting that edges can be directed, as in a connection between two vertices specifically goes from one to another. A directed graph can be used to model other types of data sets, or specific types of relations, such as people following others in an online social network.

The analyzation of these graphs using various tools or techniques to find additional data and patterns is known as Graph Mining. While there are many studies and problems in graph mining, a fundamental one is known as the *densest subgraph discovery problem* (the DSD). The aim of the DSD is that given an undirected graph *G*, you must find a subgraph *S* such that it has the highest density of all subgraphs of *G*. The density of a graph is represented as *e/v* (where *e* is the number of edges in the graph and *v* is the number of vertices in the graph). The denser a graph is, the more connected the members of that graph are. And thus, in simple terms, the DSD aims to find the most connected group of vertices within a graph. Additionally, density can also be applied to network motifs, which are small structures of vertices and edges such as shapes or cliques. In this case, it would be the number of a given motif over the number of vertices, finding the most connected group of those motifs [3]. The densest subgraph (and thus solutions to the DSD) is a notable piece of information to have for a dataset and has plenty of notable applications in real data sets including finding and filtering out fake users or identifying echo chambers in social networks, or identification of regulatory motifs in DNA or gene annotation graphs in biological data [2].

Being such a notable problem, there are of course many solutions to the DSD. However most, if not all, are serialized programs that run on the CPU. By the nature of graph mining, it is almost always required to process every vertex in these large graph datasets, which can certainly take time in a serialized program processing these one by one. By programming in parallel, a great number of vertices can be processed in concurrence, and thus save time and be more efficient. And this is where the GPU comes in. The CPU and GPU differ in their processing cores. The CPU runs on a handful of powerful processing cores that can take complex orders, while the GPU runs on many weaker processing cores that take simpler orders. So, while a powerful CPU can certainly run a taxing program efficiently, utilizing the full power by running commands in parallel across the GPU’s many cores is much more computationally efficient. Although this comes with the limitation that the GPU’s commands are much more restrictive than that of the CPU [1]. But the computational power of parallel programming on the GPU is certainly well suited for graph mining, and thus the DSD.

With the complexities of graph mining (and in this case the DSD), however, one would have a very tough time writing a program to analyze a graph only using the simple commands available to the GPU. But thanks to NVIDIA, there’s a tool to work around that. CUDA, which stands for Compute Unified Device Architecture, is a parallel computing platform and application programming interface (API) model. CUDA allows for a serialized C++ program run on the CPU to execute threads in parallel on the GPU, being able to leverage the massive computational power of the GPU as needed [1].

And that brings us to the goal of the project. Programming a solution to the DSD which utilizes the GPU through CUDA. This goal is two-fold, being both a more efficient solution to the DSD, as well as providing further research on the computational power of parallel programming on the GPU.

**Chapter 2**

**Existing Solutions**

As stated before, there are many existing solutions to the DSD. Algorithms that have been created and developed to be more and more efficient. Some algorithms will get the exact solution to the DSD, meaning these algorithms will return the densest subgraph without fail. Other algorithms will approximate the densest subgraph, finding one of the densest subgraphs, if not the densest. So, it’s worth going over some of these algorithms and covering some of the important topics to our solution they cover.

**Golberg’s maximum flow-based algorithm**

This algorithm was first developed by Andrew Golberg in 1984. The basic outline of this algorithm follows the construction of a flow network based on the given graph, where every vertex is connected to two newly added vertices s and t (the source and sink). A binary search is run on this network, maintaining an upper and lower bound on the greatest density, and tightening these bounds with every iteration until the lower bound is within the margin of being exact. The flow network and upper and lower bounds are updated using a maximum flow (or min st-cut) approach which will return the densest subgraph according to the current flow [4]. There is a lot to unpack and understand in this algorithm, but that will be discussed soon.

**Charikar’s LP-based algorithm**

(may need to replace, remove, or come back to this one)

**Greedy peeling algorithm**

This is a straightforward approximation algorithm. Given graph *G*, every iteration the algorithm will remove the vertex of lowest degree from *G*, where degree is the number of vertices a given vertex is connected to. The current subgraph of highest density is stored, and every iteration it is compared to the new subgraph. If the new subgraph is denser than the currently stored one, it replaces it. This goes on until all vertices have been removed, and thus the subgraph that had the highest density as these vertices were peeled away is the result. As is apparent, this is not a very reliable algorithm to find the densest subgraph, as it leaves a lot of possible subgraphs unchecked. However, it was found and proven that through this method, the resulting subgraph would always be *at least* half as dense as the actual densest subgraph [4]. Meaning it is a decent algorithm for finding dense subgraphs, and there is notable idea in the idea of peeling off vertices by lowest degree.

**The Greedy++ algorithm**

(This is interesting, but I don’t know if it’s entirely relevant)

**The CoreExact algorithm**

This algorithm is the most important to this paper, as it is the basis of our solution. The CoreExact algorithm is built upon Golberg’s max-flow based algorithm, but cuts down on number of iterations by pruning and peeling the given graph *G* to a more restricted subgraph that *must* contain the densest subgraph. This is done in a few steps, starting with core decomposition to find the densest k-core, then breaking that into its connected components. The densest of these components (including the density of the densest k-core) becomes the basis for the upper and lower bounds, where the lower bound becomes the density of that component, and the maximum k-value becomes the upper bound. With that, the exact algorithm is run on every connected component, and the identity of the densest subgraph is updated whenever a denser subgraph is found. This algorithm is also made to work for motifs that aren’t just edges to allow for greater versatility and use cases [3]. There is a lot to unpack here, and since this is the basis of our solution, elaborating on every piece will be handled next.

**Chapter 3**

**The Algorithm**

Our solution, based on the CoreExact algorithm, is comprised of many individual important concepts, implementations, and smaller algorithms. So we will cover everything necessary piece by piece.

**Adjacency Lists and Adjacency Matrices**

Because algorithms must be applied to actual code, there needs to be implementations of graphs programming wise. There are multiple ways to do so, but the method used for graphs in our solution is the Adjacency List. This is a fairly straightforward way of storing how vertices are connected using a 2-dimensional list (a list of lists). The first dimensional list represents every vertex in the graph, where the index matches the vertex number. The second dimension is a list of all the vertices that the represented vertex is connected to.

(insert an image here)

Additionally, since we will be allowing for the use of motifs other than regular edges, the motif structures also need to be stored. In theory, adjacency lists could be used here as well. However, we will be using adjacency matrices to do so. Adjacency matrices, in comparison to adjacency lists, work well for small structures like motifs as they make construction and representation easier in the code (which holds true for large graphs as well, but they become much more costly in memory and iteration efficiency).

As the name suggests, the adjacency matrix represents graphs through a full 2-dimensional matrix. There is a row and column for every vertex, with each intersection being filled with a 0 or 1. If the intersection has a 0, the two vertices are not connected by an edge, whereas a 1 indicates they are connected.

(insert image)

In the actual code, both are handled as 2-dimensional vectors, but their structures still reflect the adjacency list and adjacency matrix.

**K-Cores and Connected Components**

As stated with the CoreExact algorithm, we can prune down the input graph to a subgraph and break that down to save time searching. The first of these steps is finding the densest k-core.

A k-core is a graph in which every vertex is connected to at least *k* other vertices. So in a 1-core, every vertex is connected to at least one other vertex. In a 2-core, every vertex is connected to at least 2 other vertices. And so on. But within a large graph, there are likely to be subgraphs of a higher k value. To find these graphs, we can use a peeling technique like the Greedy peeling algorithm covered earlier called core decomposition. Like with the Greedy algorithm, core decomposition finds the vertex of lowest degree one at a time and removes it from the graph, where degree is the number of vertices a vertex is connected to. Using a 1-core as an example, if you remove all the vertices of degree 1 (and all subsequent vertices that have degree 1 after the removal of the other vertices), you will be left with only vertices that have degree 2 or higher, going from a 1-core to a 2-core. Going until all vertices have been removed will mean every k-core up to the highest possible *k* value will have been found. So, using this method you can record the information on all these k-core subgraphs. And as previously stated, the densest of these k-cores must contain the densest subgraph.

(may be a good to use visuals here)

With the densest k-core, we can break this down into pieces known as connected components. A connected component is a subgraph where every vertex is connected to all the others in the subgraph by *some* path. In layman’s terms, a connected component is a subgraph that has no connections to the vertices of the other connected components. Not every graph is going to have connected components, but it’s worth checking for.

(insert image here)

**Flow Networks**