**DENSEST SUBGRAPH DISCOVERY ON THE GPU**

by

Hunter G. Gareau

A Thesis

Submitted to the

Department of Computer Science

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: Guimu Guo, Ph.D., Professor, Department of Computer Science

Committee Members:

Sihan Yu, Ph.D., Professor, Department of Computer Science

Richard Rabbitz, MSc., Professor, Department of Computer Science

© 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 CoreExact Algorithm (tbd)

Chapter 4: Our Parallelized Implementation (tbd)

Chapter 5:

**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.) [1]. 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.

**The Densest Subgraph Discovery Problem**

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*.

Additionally, density can also be applied to network motifs, which are small structures of vertices and edges such as shapes or cliques.

Motif Density can be applied as edge density (when your motif type is “edge”), so it is the definition we will use. So, we can define the DSD as such:

The denser a graph is, the more connected the members of that graph are. So, in simple terms, the DSD aims to find the most connected group of vertices within a graph. 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 [1].

**The GPU And Parallel Programming With CUDA**

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 [3]. 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 [3].

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 and Related Works**

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 covering some of these algorithms to see how others have solved it before, and to introduce some important topics and ideas.

**Existing Solutions**

To start off, let’s cover a few existing algorithms that solve the DSD.

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

This algorithm was first developed by Andrew Goldberg 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 try to return a subgraph of density equal to the average of the upper and lower bounds [4]. There is a lot to unpack and understand in this algorithm, but these topics will be explored more in depth in the next chapter.

***Greedy peeling algorithm***

This is a straightforward approximation algorithm developed by Charikar (elaborate). 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***

Building on Charikar’s Greedy Peeling Algorithm is an algorithm proposed by Boob et al. (elaborate). This algorithm iterates through the Greedy algorithm *T* times, with each iteration updating the priority of each vertex so that in subsequent iterations, vertices of higher priority are kept in for longer. By running the Greedy algorithm multiple times and utilizing the results of previous iterations, denser subgraphs can be found. And while this algorithm is still an approximation, it was found that with enough iterations, the densest subgraph found would be arbitrarily close to the exact optimal densest subgraph. Meaning this relatively simple solution can find a subgraph of negligible difference from the optimal densest subgraph at worst, making for an efficient and relatively easy to implement solution to the DSD [4].

**Related Works**

Now let’s cover some algorithms that solve similar problems.

***Densest k-subgraph Approximation***

This is a notable variant of the DSD where the subgraphs being searched for are specifically of size *k* (subgraphs with *k* vertices). This can be useful information in various contexts and is also an interesting problem to dissect on its own since it essentially reduces the DSD from looking for to just In other words, you’re looking for the subgraph of size *k* with the most edges.

There are several existing solutions to this problem, but one of the major ones was developed by U. Feige, G. Kortsarz, and D. Peleg in a combinatorial approximation algorithm [5]. The basic algorithm (referred to as Algorithm A in the paper) combines three procedures for finding a dense subgraph of size *k* and returns the densest of the three. The first procedure simply takes *k/2* random edges returns the set of vertices connected by these edges, adding additional arbitrary vertices to get the set to size *k* if needed. This procedure provides a baseline, always returning a subgraph of density ≥ 1. The following 2 procedures act as Greedy approximations to try and find an even denser subgraph. The first of the two sorts the vertices by degree, and takes the k/2 vertices of highest degree into the subset, then resorts the remaining vertices by how many neighbors they have in the initial subset, and adding the last k/2 vertices from the top of that ranking. The second of these procedures constructs a subgraph for every vertex *v* in the graph, being constructed by ranking how many 2-step paths each vertex has to *v* and then ranking the neighbors of *v* by how many of those vertices they’re connected to, followed by taking the k/2 vertices of the highest degree connected to those vertices. The union of this set and the 2-step neighbors serves as the resulting subgraph, adding in arbitrary vertices if the result does not reach k. As stated before, the result of these three procedures with the greatest density is taken as the result of the algorithm. This algorithm is of accuracy . In the same paper, they dive into ways to approximate even closer, and other algorithms have been made to approximate the densest k-subgraph which have a higher accuracy, but this algorithm served as the first notable solution.

***DSD for Directed Graphs***

As was mentioned earlier, the definition of density we utilize only applies to undirected graphs, as that definition does not take directionality into account. A definition for density in directed graphs was proposed by Kannan and Vinay in 1999 [6], and is still in use today. It is defined as follows:

So in layman’s terms, directed graph density represents how connected one set of vertices is to another. And as follows, the Directed Densest Subgraph Problem (DDSP) aims to find sets such that *d(S, T)* is maximized.

There are multiple solutions to this problem as one may expect, and one such algorithm developed by Charikar will find the exact solution. This solution is a Relaxed Linear Programming problem based on the value of *|S| / |T|* (referred to in the paper as *c*). As proven in [7], the optimal value of the linear programming problem on c is equivalent to the optimal directed density, where the optimal sets *S* and *T* can be computed from the results of the LP.

***Optimal Quasi-clique Problem***

The definition of density most widely used is the one covered in Definition 1. However, some have put forward different definitions to gather different results. One such definition is called Edge-Surplus:

This definition is flexible, as the definition of *g*, *h*, and are determined for the specific case. But this framework sets up the problem such that it favors more edges and penalizes more vertices (hence why is a positive value, and why is negative).

This definition can be used to evaluate the DSD as normal, but in the case of quasi-clique: , , and These definitions favor subgraphs that are tighter knit and have a small diameter (subgraphs with longest paths that are smaller). This differs from the normal density definition which does not distinguish graph size. And as follows, the Optimal Quasi-clique Problem (OQP) aims to find the subgraph that maximizes the value of this function. One such solution to OQP is based on the Greedy peeling algorithm for the DSD. The main differences lie in that obviously this algorithm checks for quasi-clique value rather than standard edge density, and that they increase efficiency by keeping lists of all possible degree values, updating them as vertices are removed and using them to decide which vertex to remove next.

Now while these are all very interesting algorithms and problems to discuss, let’s move on to the most important algorithm of our project.

**Chapter 3**

**The CoreExact Algorithm**

Our program is based on the CoreExact algorithm [2]. We will be parallelizing the algorithm in a C++/CUDA program, but to understand the implementation, we should first explain CoreExact. There are a lot of concepts and smaller algorithms that make up CoreExact though, so we will go into each of these before putting it all together.

**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 program (and in the CoreExact program) is the Adjacency List. This is a 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. A representation of this is shown in Figure 1.



*Figure 1.* Adjacency List Representation [5].

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. A representation of this is shown in Figure 2.

A diagram of a network with numbers and circles

AI-generated content may be incorrect.

*Figure 2*. Adjacency Matrix Representation [6].

In the actual code, both adjacency lists and matrices are handled as 2-dimensional vectors, but their structures still reflect what has been outlined above.

**K-Cores and Connected Components**

A major aspect of the CoreExact methodology is that 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.

Figure 3 outlines an example of a graph which can be broken down into a 1-core, 2-core, and 3-core. The orange vertices are in the 1-core since they all have a degree of 1 (they’re only connected to 1 vertex) and are removed at the first stage. The blue vertices are in the 2-core since they only have a degree of 2 after the orange vertices’ removal. This leaves the red vertices, which are all connected to 3 vertices each, leaving a 3-core.



*Figure 3*. Graph with outline of 1-core, 2-core, and 3-core [7].

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 Figure 4 is an example of a graph of connected components, where the graph can be broken into 3 disjointed subgraphs *{V1, V2, V3, V4, V5, V6}, {V7, V8, V9}, and {V10, V11, V12}*. 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.



*Figure 4*. Connected components example [8].

**Flow Networks and Min st-cuts**

One of the most important pieces of CoreExact is the usage of Goldberg’s maximum flow algorithm. A key aspect of that algorithm is a constantly updating flow network using the next best guess for greatest density, and then using min-st cut to return the densest subgraph based on that density. So, to understand how Goldberg’s algorithm works, it’s important to understand what flow networks are, and what a minimum st-cut is.

A flow network is a special type of directed graph where every edge has two values: capacity and flow. These are abstract numerical values that represent an amount of “something” passing through the edge in the direction it faces. Water through a pipe is an easy way to look at this, since water flows through a pipe in a certain direction. Capacity represents the maximum amount of this “something” that can pass through the edge, while flow is an amount greater than or equal to 0 that is less than or equal to the capacity, essentially representing how much of this “something” is passing through the edge. This is often represented as *x/y*, where *x* is the flow and *y* is the capacity. Using the water pipe example, a pipe may be able to sustain up to 5 liters of water at any moment, but the amount of water passing through may be anywhere from 0 to 5 liters. Flow networks have two special vertices called the source and the sink, usually referred to as *s* and *t* respectively. The source is where all flow starts and comes from, and the sink is where all the flow ends up. Another important detail is that flow follows the rules of conservation, where the amount of flow going into any given vertex must be equal to the amount coming out of it (unless they are the source or sink).



*Figure 5*. Flow Network example [9].

Shown above in Figure 5 is an example of a flow network. As can be seen, every edge has a flow/capacity pair, with the flow ranging from 0 to the capacity of that edge. We can see that flow only comes out of source s, and flows into sink t. Additionally, flow conservation can be examined here. As an example, Vertex A receives 8 flow (5 from s and 3 from D), and outputs 8 flow (5 to B and 3 to C). This rule is held up in the other vertices B, C, and D as well.

An st-cut is a division of the flow network into 2 subgraphs *S* and *T*, where *S* contains the source, and *T* contains the sink. This cut is made by removing edges such that there is no remaining connection between the two subgraphs. Cut capacity is the sum of capacities of the removed edges, but an important note is that only the capacity of edges that flow into *T* are counted in cut capacity. This can be examined in Figure 6. A cut of the network is being depicted such that the edges connecting 0 to 2, 2 to 1, and 1 to 3 are removed from the graph. This leaves two subgraphs, *S = {s, 0, 1}* and *T = {2, 3, t}*. We can calculate the cut capacity to be 5, since we ignore the capacity of edge 2 to 1 as it flows into *S*, leaving capacities of 2 and 3.

A screenshot of a computer

AI-generated content may be incorrect.

*Figure 6*. st-cut.

The minimum st-cut is the st-cut where cut capacity is minimized. Importantly, it’s been found that the value of this cut is equal to the maximum flow of the network, and so the problems can be used interchangeably. Using the same graph, we can find that the minimum st-cut is pictures in Figure 7, where we are left with *S = {s, 0}* and *T = {1, 2, 3, t}.* The cut capacity comes out to be 4, as the edge from 1 to 0 is excluded due to flowing into *S*, so leaving capacities of 2 and 2. If you examine the example graph further, you will find no st-cut with a cut capacity less than 4, which is why Figure 7 depicts the minimum st-cut.

A screenshot of a computer

AI-generated content may be incorrect.

*Figure 7*. Minimum st-cut.

In this algorithm, we use the Edmonds-Karp algorithm to find max flow, and therefore the min st-cut. Edmonds-Karp uses a breadth first search to find the shortest path from *s* to *t*, pass the max flow it can along this path, update the flow network to include this flow, then repeat until no more paths can be found. This results in a network of maximum flow, and by taking the st-cut of this graph, *S* (which will contain all vertices with flow running through them), will be the subgraph to check.

(maybe include a visual?)

**An in Depth Look at Goldberg’s Algorithm**

Now that we’ve explained the necessary pieces, we can do an in-depth explanation of how Goldberg’s maximum flow algorithm works in CoreExact.

To start off, lower and upper bounds *l* and *u* are declared, and α is set as the average of *l* and *u.* Then, a specifically designed flow network that is based on any input graph is utilized. In this network, every edge (*u,v*) is replaced by a pair of directed edges, one from *u* to *v* and one from *v* to *u*, each with capacity 1. Two additional vertices are added to the network to be the source and sink (*s* and *t*). There is an edge added for every vertex *n* in the graph from *s* to *n* where the capacity is equal to the motif degree of *n*. There is also an edge added for every vertex *n* in the graph from *n* to *t* where the capacity is equal to α times the motif size. As a note, in Goldberg’s original algorithm, it uses edge degree and just α. But CoreExact has adjusted those values to work with motifs other than edges. With this graph, we can take the minimum st-cut to see if there exists a subgraph of density α or higher.

(cover the math?)

(cover “margin of error”, the α value checker)

**The Full Algorithm**

Now that we’ve covered the individual aspects, let’s review how this algorithm works. Before the algorithm begins, a graph and motif type are selected as input. To start off, core decomposition is run on the graph to find the k-cores and store the information on the densest of them. The densest k-core is then broken into connected components, and the highest density is found between the k-core and its connected components. The densest of these is saved as the current densest subgraph, and we get the upper and lower bounds, *l* and *u*, from them (where *u* = the *k* value of the k-core and *l* = the greatest density). At this point, the binary search for a denser subgraph begins, being run on each of the connected components. The next best guess for highest density is used to construct the flow network. The minimum st-cut is taken of this flow network, returning *S*. We then check if *S = {s}* or not. If *S = {s}* is true, a subgraph of density greater than or equal to α was not found, and thus the upper bound is now set as α. If *S = {s}* is not true, then a subgraph of density greater than or equal to α was found, and the density of that subgraph now equals the lower bound and that subgraph is saved as the current densest. This runs until the difference between the upper and lower bound is within the margin of error. Once that happens, we either move onto the next connected component and redo the previous steps, or we are done, and we have found the densest subgraph.

With all of this in mind, we can cover the important aspects of our implementation of the CoreExact algorithm.

**Chapter 4**

**Our Parallelized Implementation**