OUT-OF-CORE GRAPH COLORING ALGORITHM

by

Yiqian Liu, B.A., M.A.

A thesis submitted to the Graduate Council of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Computer Science
August 2020

ACKNOWLEDGEMENTS

When I joined the Computer Science Department at Texas State University, I only had little experience in computer science and research. But now, I am having completed a master thesis. I have a wonderful time in San Marcos within the two-year master's study. I would like to take this opportunity to thank many of the people that I met here.

First and foremost, I would like to thank my advisor Martin Burtscher. He is supportive and always provides valuable feedback. His passion for research and working with students motivates me to be more interested in research. I would also like to thank my thesis committee: Qijun Gu and Tanzima Islam. I enjoyed and learned a lot from our discussion about and beyond the thesis.

Additionally, I would like to thank my family, in particular, my parents and my husband for always supporting me and for their love.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Out-of-core algorithms can process data sets that are too large to fit entirely into the computer's main memory. This thesis develops an out-of-core algorithm for graph coloring. It dynamically partitions the graph into subgraphs, processes them in sequence, and records the color information needed by later subgraphs in a dense format. The algorithm is guaranteed to produce the same coloring as the first-fit in-core algorithm. It employs a new method to compactly record information and automatically resizes the associated data structure to save memory. As there are no pre-existing out-of-core graph coloring codes, the implementation can only be compared to leading in-core graph coloring codes. Based on the geometric mean over 18 graphs from various domains, JP-D1 is 25% faster and uses 13% fewer colors. FirstFit and Boost both use the same number of colors as the presented implementation, but FirstFit is 4 times faster whereas Boost is 6 times slower.

1. INTRODUCTION

Out-of-core algorithms, or external algorithms, refer to processing large data sets that are too massive to fit completely inside the computer's internal memory (Vitter, 2001). As computers usually contain a memory hierarchy, the speed to access internal memory and external memory differs greatly. The major performance bottleneck of the out-of-core algorithm is the input/output (or I/O) communication between the internal memory and the external memory.

This thesis explores solutions to this problem for graph coloring. Formally, a (vertex)-coloring of an undirected graph $G = (V, E)$ is an assignment of a color $v.color$ to each vertex $v \in V$ such that, for every edge $(u, v) \in E$, $u.color \neq v.color$ holds (Hasenplaugh et al., 2014). In addition, the goal of graph coloring is to use as few colors as possible.

Graph coloring problems arise in many practical applications such as compiler register allocation, constructing timetables, and taxi scheduling (Lewis, 2015). The rapid growth of the Internet and the recent advances in information technology results in large scale data sets, which can be modeled as graphs (Lin et al., 2017). The practical importance and the growth of massive graphs brings attention to the out-of-core graph coloring algorithm.

Since graph coloring is NP-hard (Zuckerman, 2006), several heuristic algorithms have been developed. The greedy algorithm is one of them. It iterates over vertices in a certain order and colors the current vertex with the best available color (*i.e.*, the smallest color that is not assigned to its already colored neighbors). In practice, the order in which the vertices are colored in the greedy algorithm affects the number of colors needed (Hasenplaugh et al., 2014).

To reduce the number of colors, several ordering heuristic have been proposed (Alabandi et al., 2020), including first-fit (FF), where the vertices are colored in the order in which they are given in the input, random (R), where the vertices are colored in random order, and largest degree first (LDF), where the vertices with larger degrees are colored first.

The massive graph is most easily partitioned into subgraphs (to fit into the CPU's main memory) according to the order in which the vertices appear in the vertex set. Consequently, I use the greedy algorithm with first-fit (FF) ordering. It assigns an ID to each vertex based on its location in the vertex set (*i.e.*, the first vertex in the set has ID 0, the second vertex in the set has ID 1, and so on). A lower ID represents a higher priority.

Each subgraph may have edges that connect to previous subgraphs or later subgraphs. Hence, I need a good data structure to record this information and pass it to the later subgraphs such that the global result is correct (*i.e.*, each vertex is assigned the same color as when processing the graph as a whole). For performance reasons, I also need to minimize I/O, that is, this information should be stored as compactly as possible and operating on it should be efficient.

The contributions of my thesis are as follows.

- I developed an out-of-core graph coloring algorithm that can process graphs that are too large to fit in a user-defined memory size.

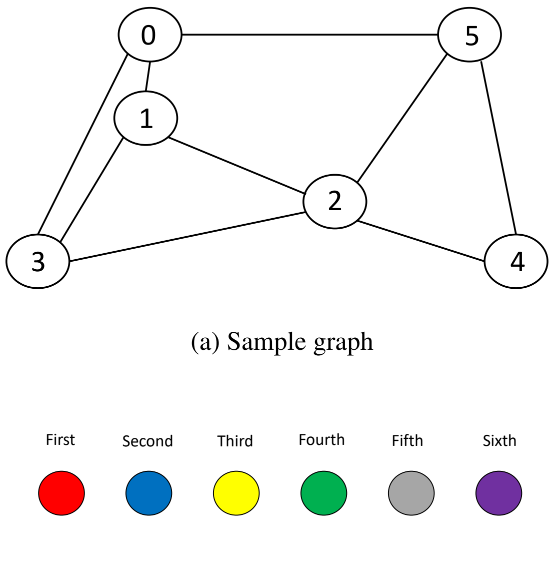
- I created a data structure that minimizes the storage space and supports fast insertion and searching.

- I evaluated the performance of different parameter values (*e.g.*, the number of bit vectors associated with the colors, the ratio of the information size to the maximum information size, the ratio of the maximum graph size to maximum total size) to tune my implementation and to gain insight.

2. BACKGROUND

In graph coloring, the available colors have a particular order (the first color, second color, *etc.*). The first color is the best color and has the highest priority. For a vertex v, if at least one of its neighbors has already been assigned the first color, the second color becomes the best available color. If that color is also already used by at least one neighbor, the third color becomes the best available color and so on.

For a graph $G = (V, E)$, the degree of a vertex v is the number of its neighbors, and the degree of G is the largest degree among all vertices. In the greedy graph coloring algorithm, the upper bound on the number of colors needed is one more than the degree of G (Hasenplaugh et al., 2014). For a vertex in the graph, the range of its available colors is one more than the degree of the vertex. Figure 2.1 (a) shows an example graph to describe the out-of-core coloring algorithm I developed. Figure 2.1 (b) presents the color order that is used in the illustration.

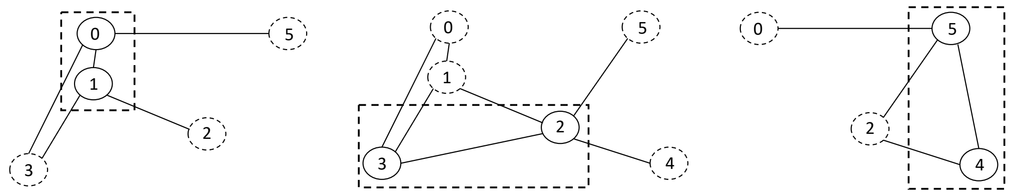


(a) Sample graph

(b) Color order

Figure 2.1: Example graph and assumed priority among the colors

The sample graph has 6 vertices and I assumed only subgraphs with 2 vertices fit in the memory. Figure 2.2 shows each subgraph (the vertices and edges that are not connected to the subgraph are not presented). As the adjacency list of each vertex contains all neighbors, even though each subgraph only has 2 vertices, it still holds the edges that connect to the vertices that are not in the current subgraph.



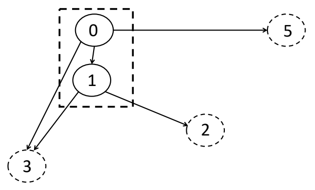
(a) First subgraph (b) Second subgraph (c) Third subgraph

Figure 2.2: Subgraphs after paritition

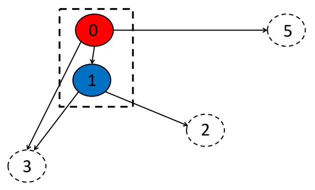
For each subgraph, the first step is to transform the subgraph into a directed acyclic graph (DAG). The direction of the edge is from a smaller vertex ID to a larger vertex ID. The first subgraph is converted as shown in Figure 2.3 (a). A vertex with a smaller ID has a higher priority. Hence, the vertex with ID 0 is colored first. For each vertex, only the color of its parents (higher-priority neighbors) matter. Vertex 0 has no parent, so the first color is the best color for it. Vertex 1 has a parent (vertex 0) colored with the first color, so its best color is the second color. Figure 2.3 (b) presents the first subgraph after coloring.

In the first subgraph, vertex 0 and vertex 1 both have edges that connect to later subgraphs. The color and ID of both vertices are, therefore, required by later subgraphs. This information, which is shown in Figure 2.3 (c), is recorded and must be passed to later subgraphs.

Each subgraph repeats the above three steps: 1) convert the subgraph into a DAG, 2) color each vertex, and 3) record the color and ID of each vertex that has edges connecting



(a) DAG representation



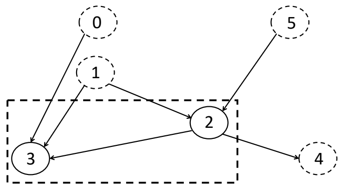
(b) After coloring

Color	Vertex ID
First	{0}
Second	{1}

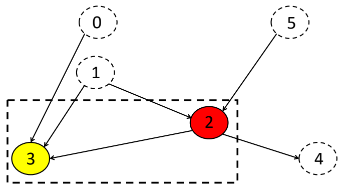
(c) Recorded information

Figure 2.3: Coloring the first subgraph

to later subgraphs. Figure 2.4 and Figure 2.5 show the steps to color the second and third subgraph, respectively. All vertices are colored and no adjacent vertices have the same color after processing every subgraph. Figure 2.6 shows the final result.



(a) DAG representation

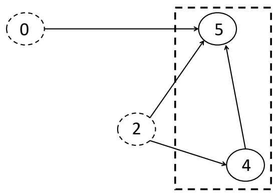


(b) After coloring

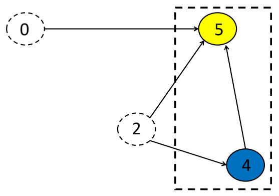
Color	Vertex ID
First	{0, 2}
Second	{1}
Third	{3}

(c) Recorded information

Figure 2.4: Coloring the second subgraph



(a) DAG representation



(b) After coloring

Color	Vertex ID
First	{0, 2}
Second	{1}
Third	{3}

(c) Recorded information

Figure 2.5: Coloring the third subgraph

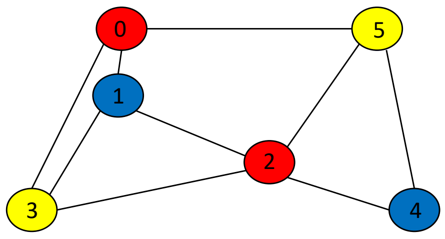


Figure 2.6: Final coloring result

3. IMPLEMENTATION

3.1 Dynamic Partitioning of the Graph

I assume the graph is stored in CSR sparse matrix format to store the graph in 2 arrays (Chen et al., 2017), which is a dense and frequently used graph representation.

My algorithm first partitions the graph by determining a cut point in the two CSR arrays. This cut point becomes the starting point of the next iteration. The goal is to find the cut point quickly and partition the graph to make sure the local subgraph and the recorded information fit in the given memory size.

A binary-search-based approach is used to make this step efficient. It first calculates a possible cut point according to the average degree of the graph. Then, it uses binary search to refine the cut point such that the local subgraph is as large as possible but smaller than the available size while not cutting a vertex's adjacency list in the middle.

As the method described in Chapter 2 requires space to store information about vertices that have neighbors in later subgraphs, part of the memory is assigned to this information and the other part is assigned to the local subgraph and local result. A global variable G is used to express the initial ratio of the maximum subgraph size to maximum memory size. The default value of G is 0.5, meaning that half of the memory is assigned to the local subgraph (and local result) and the other half to storing color information about earlier subgraphs.

Considering that there are many different graphs, a static G value may not work in all cases. For example, a dense graph with a large average degree likely requires to store relatively more information about previous subgraphs than a sparse graph. Hence, I change the value of G dynamically according to the current information size and local subgraph size. In other words, the maximum subgraph size is shrunk and expanded as the current information size changes. A global threshold R is introduced to indicate if the maximum graph size must be recalculated.

To boost performance, recorded color information that is no longer required for later subgraphs is only deleted whenever the ratio of the current information size to the maximum information size exceeds R. If the ratio still exceeds R after the deletion, the maximum graph size is shrunk to leave more available memory to store color information.

I assume the information size in the next iteration is at most twice the current information size for most graphs. Hence, the ratio of the information size to maximum information size is not likely to exceed the threshold R in the next iteration if this ratio is currently under $R/2$. In fact, I increase the allowed maximum graph size when the ratio is less than $R/2$. This enables larger subgraphs going forward, which reduce the number of iterations and require less frequent I/O.

3.2 Bit Vectors

Memory is a scarce commodity in out-of-core algorithms. To save memory when storing sets of vertices with the same color, I use a a bit vector of size N to represent a set of N integers. Figure 3.1 depicts such a structure. A vertex ID v can be represented as v/N and $v\%N$ which is recorded in the bit vector. This bitmap covers the N vertices with offset 0 through $N-1$. If the k^{th} bit is true, the vertex whose ID with offset k is in the set.

A set with a range of N integers requires up to $4*N$ bytes in memory, assuming each 4-byte integer is listed explicitly. The bit-vector version requires $N/8$ bytes, which is up

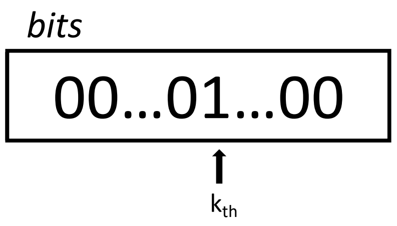
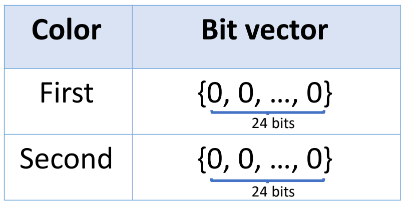


Figure 3.1: The base value and its associated bit vector

to 32 times smaller for densely populated sets. To search if vertex *v* is in the set, we first need to check if a chunk with the value v/N exists. If it does not, return false. Otherwise, check if bit $v\%N$ in that chunk is true. To insert a vertex *v*, check if a chunk with the value v/N exists. If it does, set bit $v\%N$. If it does not, create a new chunk and set bit $v\%N$ in it. Hence, the bit-vector representation may comprise multiple chunks.

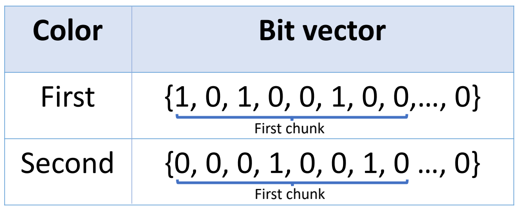
This representation is very helpful for recording the vertices that are colored with the first few colors as these colors are typically used by most vertices in the graph. For each of these colors, I use such bit vectors to indicate whether a vertex has this color. There is only one chunk per color as *N* equals the number of vertices. Each bit in the bit vector represents a vertex and is initialized to false. It is set to true when inserting the vertex ID (*e.g.*, the best color for this vertex is the color associated with this bit vector and the vertex has a neighbor in later subgraphs). I use a global parameter *C* to specify the number of colors that have such an associated bit vector. The default value of *C* is 2 because the majority of the vertices use either the first or the second color in many graphs of practical interest ((later results will show that the majority of the vertices use the first two colors)).

Here follows an example of a graph with 24 vertices that is partitioned into 3 subgraphs to describe how this bit vector is used to record the vertex ID. I assume the first subgraphs processes vertices 0 through 7, the second subgraphs 8 through 15, and the last subgraphs 16 through 23. As $C = 2$, the first color and the second color both have an associated bit vector. Each bit vector consists of 24 bits to represent each vertex in the graph. Every bit is initialized to false as presented in Figure 3.2(a). The first subgraph is processed and vertices 0, 2, and 5 are colored with the first color and vertices 3 and 6 are colored with the second color. Hence, bits 0, 2, and 5 of the first bit vector and bits 3 and 6 of the second bit vector are set to true as shown in Figure 3.2(b). Then the second subgraphs is processed. Vertices 10, 13, and 14 are colored with the first color and vertices 9, 12, and 15 are colored with the second color. The bit vectors are updated as shown in Figure 3.2(c). The last subgraph does not need to record any such information because the bit vectors are only used to record the colors of vertices needed in later subgraphs.



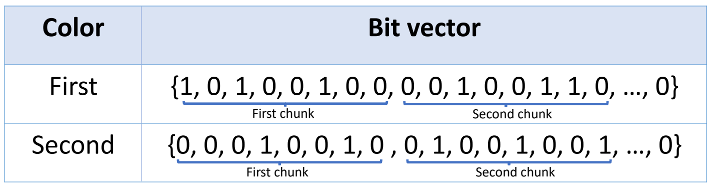
Color	Bit vector
First	$\{\underbrace{0, 0, \ldots, 0}_{\text{24 bits}}\}$
Second	$\{\underbrace{0, 0, \ldots, 0}_{\text{24 bits}}\}$

(a) Initializing the bit vectors



Color	Bit vector
First	$\{\underbrace{1, 0, 1, 0, 0, 1, 0, 0}_{\text{First chunk}}, \ldots, 0\}$
Second	$\{\underbrace{0, 0, 0, 1, 0, 0, 1, 0}_{\text{First chunk}} \ldots, 0\}$

(b) After processing the first subgraph



Color	Bit vector
First	$\{\underbrace{1, 0, 1, 0, 0, 1, 0, 0}_{\text{First chunk}}, \underbrace{0, 0, 1, 0, 0, 1, 1, 0}_{\text{Second chunk}}, \ldots, 0\}$
Second	$\{\underbrace{0, 0, 0, 1, 0, 0, 1, 0}_{\text{First chunk}}, \underbrace{0, 1, 0, 0, 1, 0, 0, 1}_{\text{Second chunk}}, \ldots, 0\}$

(c) After processing the second subgraph

Figure 3.2: Bit vectors for the first two colors

I only use this data structure for vertices that are colored with the first or second color as those usually result in densely populated sets. The colors of the remaining vertices are recorded in a different data structure. Since the later colors are used much less frequently, employing bit vectors for them would be inefficient as they would mostly contain zeros.

3.3 Compressed Lists

Recording which vertex has which color can easily be achieved by storing each vertex's color in an array that is indexed by the vertex ID. Each array element would have to be large enough to record any possible color, so probably an 8- or 16-bit integer. This array would likely be too large for an out-of-core implementation. Instead, I opted (1) to only hold the color information in memory for vertices that have neighbors in later subgraphs and (2) to get close to storing just a single bit of information per such vertex. The resulting data structure is described in this section.

Most graphs, even very large sparse graphs, usually only require a small number of colors. Hence, a possible way to efficiently record which vertex has which color is to record the IDs of all vertices of a given color in a data structure. This way, the color value is implicit and does not need to be stored. As described in Chapter 3.2, the first and second colors both use a bit vector to record the vertex IDs. For efficiency reasons, higher colors require another data structure.

First, I thought a binary search tree might be a good choice. However, this thesis implements the FF order, meaning the vertex IDs are inserted into the data structure in increasing order. It leads the binary search tree to become a linked list. Hence, I created a compressed lists data structure for each color to store the vertex IDs, which is presented in Figure 3.3. These lists allow for dense storage, support fast insertion, and can be searched relatively quickly.

Each subgraph has its own color list and dynamic arrays. However, the vector of chunks is shared by all subgraphs. The color list of each subgraph is a list of pointers to the dynamic arrays. The dynamic array entries are integers specifying the index of the element (chunk) in the vector to which they refer.

The size of the color list is determined by the highest color number in the subgraph, *i.e.*, it is typically very small. The vector of chunks is dynamically increased and decreased (see next section). The size of the dynamic array is n/N, where n is the number of vertices in the subgraph. It is indexed by the value V/N where V is the vertex ID. Each vertex has a deterministic position in the dynamic array. In other words, the dynamic array can be accessed directly through the vertex ID without any search. Each chunk in the vector has a bit vector for the vertex ID offset and can, therefore, also be accessed directly without search. The chunks also contain an integer to indicate the maximum neighbor ID in the range of the vertices that are represented by the bit vector. This information makes it possible to recycle the chunk as soon as it is no longer needed to keep the data structure small.

Here follows an example of a graph $G = (V, E)$ to explain the compressed lists in more detail. The graph is partitioned so that the number of vertices and edges in each subgraph does not exceed the maximum allowed size and a vertex's adjacency list is not cut in the middle. To simplify this example, I assume the graph is partitioned into 2 subgraphs and each subgraph has $V/2$ vertices.

As presented in Figure 3.3, the first subgraph and second subgraph have their associated color list and dynamic arrays. They share the vector of chunks. The color list of every subgraph is initially empty to save memory. Whenever a color is added, the corresponding dynamic array is allocated with a size of $V/(2*N)$ and filled with -1. Any unused colors have a NULL pointer and no dynamic array is allocated, such as the pointer of the fifth color in the second subgraph.

To insert a vertex ID, it locates the corresponding entry in the color list of the current subgraph and checks the pointer. If it is NULL, it allocates and initializes the dynamic array. Then it accesses the dynamic array according to the vertex ID to get the index. If the element at this index is -1 (*e.g.*, this vertex has no associated chunk in the vector), it sets the index to the current size of the vector and adds a new chunk to the vector. Also, it sets the associated bit in the chunk as described in Chapter 3.2. If the index is not -1, it accesses the vector with this index and sets the bit in the associated chunk to true.

To search for the color of a vertex, it iterates over the C bit vectors and then over every dynamic array in the associated subgraph (which is determined by the vertex ID) until finding the vertex. For example, assuming vertex j is in the first subgraph with $j/N = 3$. To search for its color using the data structure depicted in Figure 3.3 and assuming it did not find it in the C bit vectors, it starts with the dynamic array of the third color in the first subgraph. Accessing this dynamic array yields an index of -1 in position 3. This means the vertex ID is not found. So it checks position 3 in the next dynamic array, where the index is again -1. It continues to check position 3 in the dynamic array of the fifth color. Here, the index is 1. So it accesses the vector with this index. The vertex ID is found if the associated bit in the selected chunk is true.

Insertion and searching are efficient as accessing the dynamic arrays and the vector of chunks take $O(1)$ time. The search operation takes $O(k)$ time where k is the color number of the vertex because we need to check one color after the other. Note that this is still

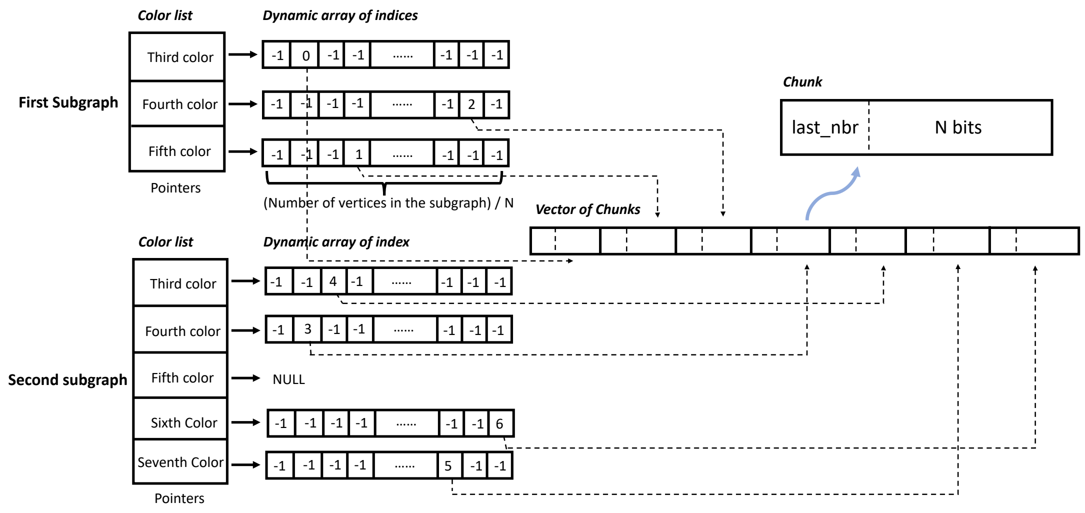


Figure 3.3: Compressed lists to record the vertex ID

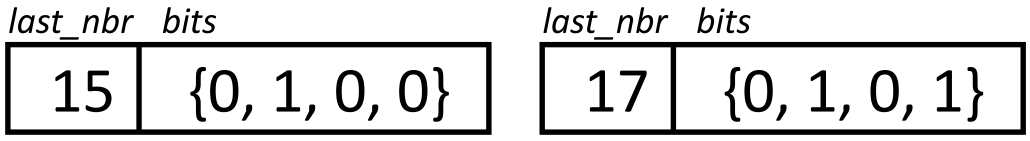
efficient as most vertices are colored with a low color number. The vector of chunks is implemented by the vector in the standard template library. Adding an element at the end of the vector is amortized $O(1)$. Hence, the insertion operation is $O(1)$.

This compressed list also saves memory. The dynamic array is only allocated when necessary (*i.e.*, when the pointer in the color list is NULL and a vertex ID is going to be inserted). The size of the dynamic array is small, it is N times smaller than the range of the vertices in a subgraph. Also, the vector is dynamically resized and its unused elements are recycled (see below).

3.4 Resizing the Vector

The vector of chunks should be as small as possible. Hence, it needs to be resized by deleting unnecessary chunks (*i.e.*, chunks that are not required by later subgraphs) to save memory.

A field that records the maximum neighbor vertex ID is included in each chunk to indicate when this bit vector is safe to be deleted. The structure of each chunk in the vector is shown in Figure 3.4. Assume the maximum neighbor ID is currently 15 for the vertices 0 to 3. Assume further that vertex 3 is inserted into the bit vector and its maximum neighbor ID is 17. Since 17 is larger than 15, we need to update the value of last_nbr with the larger neighbor ID. The structure after the insertion is presented in Figure 3.4.



(a) Adding a field of last_nbr (b) Inserting vertex 3 which has vertex 17 as neighbor

Figure 3.4: Updating the last_nbr

If the maximum neighbor ID is less than the last vertex ID of the current subgraph, then the chunk is no longer required in the following iterations. The resizing of the vector is implemented by moving every chunk that is not deleted to a new vector. This is implemented by iterating over the dynamic arrays to access the vector chunks and updating the indices in the dynamic arrays accordingly. Given that this process is expensive, I only resize the vector when the information is about to exceed the maximum allowed information size. The resizing function is called when the ratio of the information size to the maximum information size exceeds the threshold R.

4. RELATED WORK

4.1 Out-of-core Algorithms

Several techniques have been developed to process large-scale graphs on CPUs on a single machine as well as on clusters. In 2001, Jeffrey Vitter surveyed the design and analysis of out-of-core algorithms and data structures to exploit locality in order to reduce the I/O costs between disk and main memory on CPUs (Vitter, 2001). His paper discusses techniques for sorting, operations on matrices (such as matrix multiplication), geometric data, graphs, dictionary lookup, and range searching. It also considers two important data structures that are based on extensible hashing and B-trees. Similarly, I designed two compact data structures that are useful for out-of-core graph coloring.

A disk-based system, GraphChi, was proposed in 2012. It uses parallel sliding windows to update the graph on disk in order to reduce the I/O costs (Kyrola et al., 2012). Sliding windows is a good method for some algorithm but makes it hard to guarantee a correct result for graph coloring.

In 2014, a graph data offloading technique using non-volatile memory devices (NVMs) was introduced to augment the hybrid BFS (Breadth-First Search) algorithm (Iwabuchi et al., 2014). PrefEdge, a prefetcher for graph algorithms that combines a judicious distribution of graph state between main memory and solid state drives (SSDs) with a read-ahead algorithm to prefetch needed data in parallel, was developed in 2014 for mining large graphs (Yoneki et al., 2014).

MOSAIC, a graph processing engine for a single machine was introduced in 2017. It enables graph analysis on one trillion edges, employs a new data structure - Hilbertordered tiles - for locality, load balancing, and compression, and proposes a hybrid computation and execution model that efficiently executes both vertex- and edge-centric operations (Maass et al., 2017). These techniques all target fast memory devices (*e.g.*, SSDs and NVMs).

Since GPUs typically have substantially less main memory than CPUs, out-of-core algorithms are of particular interest in that domain. A fast and scaleable graph processing method, GTS, that handles even 64 billion edges very efficiently on GPUs was proposed in 2016. This method stores graphs in PCI-E SSDs (Kim et al., 2016). Even though GTS is able to process large-scale graphs, it still requires SSDs just like the above CPU methods.

TOTEM was the only system for processing large-scale graphs on GPUs before GTS. It provides an environment to implement graph algorithms on hybrid CPU/GPU platforms (Gharaibeh et al., 2013). Even though TOTEM is able to process large-scale graphs on GPUs, it often only processes a small part of the graph on the GPU and may underutilize the computational power of GPUs.

The general graph processing platform Garaph was designed to process large-scale graphs with better parallelism on the CPU and GPU of a single machine (Ma et al., 2017). It employs a replication factor to maximize GPU utilization and edge-based partitioning to improve the work balance on the CPU. It also proposes an adaptive scheduling mechanism to exploit the overlap of the two devices (GPU and CPU) and multistream scheduling for data transfer and GPU kernel execution overlap.

Graphie, a large-scale graph traversing system for a single GPU was developed to reduce the communication between CPU and GPU by storing the vertex attribute data in the GPU memory and stream edge data synchronously to the GPU for processing (Han et al., 2017).

4.2 Graph Coloring

As mentioned in the introduction, the order in which the greedy coloring algorithm colors the vertices matters. There are many studies on serial ordering heuristic. A 2014 paper evaluates 6 of the most popular heuristics: 1) first-fit ordering (FF), where the vertices are colored in the order in which they appear in the vertex set, 2) random ordering (R), where the vertices are colored in random order, 3) largest-degree-first ordering (LDF), where the vertices with larger degrees are colored first, 4) smallest-degree-last ordering (SDL), where the vertices with the smallest degree are successively removed from the graph, the modified graph is colored using the LDF heuristic, and finally the removed vertices are reinserted and colored, 5) saturation-degree ordering (SD), where the vertices whose colored neighbors have the largest number of unique colors are colored first (using the vertex degree as a tie breaker), and 6) incidence-degree ordering (ID), where the vertices with the largest number of colored neighbors are colored first irrespective of the number of unique colors (using the vertex degree as a tie breaker) (Hasenplaugh et al., 2014). This study highlights the tradeoff between coloring quality and runtime. I use FF because it is both fast and yields a reasonable color quality while fitting my partitioning strategy well.

ColPack is a package comprising of implementations for a variety of graph coloring and related problems, including general graph coloring, bipartite graph one-sided coloring, and bipartite graph bicoloring (Gebremedhin et al., 2013). It includes a greedy algorithm with various ordering techniques for general graph coloring such as largest degree first, smallest degree first, *etc.* (ColPack, 2019).

Several parallel strategies have been proposed to promote the performance of graph coloring. One approach to parallel graph coloring uses independent sets. Luby's parallel maximal independent set algorithm is one of them (Luby, 1986). It finds the maximal independent set in parallel and all vertices in the independent set are assigned the same color. Jones and Plassmann proposed another parallel graph coloring algorithm that chooses a random number for each vertex and assigns each vertex to a processor (Jones and Plassmann, 1993). Each vertex is colored with the best color available to it.

The Parallel Boost Graph Library is an extension of the Boost Graph Library (BGL) for parallel and distributed computing (Boost, 2009). It implements graph coloring by iterating over the vertices once and selecting the lowest-numbered available color. The number of colors used in Boost is related to the sequential order of the vertices in the serial cases. The distributed version produces a different number of colors depending on the ordering, the distribution of the vertices, and the number of parallel processes cooperating to perform the coloring.

Graph coloring on GPUs has been widely researched in recent years. Grosset *et al.* proposed the first GPU graph coloring algorithm (Grosset et al., 2011). Their work is based on the Gebremdhin-Manne algorithm that has 3 phases: optimistic coloring, conflict detection, and conflict resolution to color the graph for a shared memory computation model. Naumov *et al.* developed a ”csrcolor” implementation using the cuSPARSE library (Naumov et al., 2015). It operates on the graph in CSR format, which is the same format I am using in this thesis. Che *et al.* observed that a static work allocation runs into load imbalance so that they use the largest-degree-first for early iterations followed by a randomized strategy (Che et al., 2015). Merill also proposed a load balancing optimization that maps the workload of a vertex to a thread, wrap, or block depending on the size of the vertex's adjacency list (Merrill et al., 2012).

Chen *et al.* (Chen et al., 2017) proposed two graph coloring algorithms on GPUs with multiple optimizations such as a bitmap to reduce the memory footprint and Merill's load balancing strategy. The first algorithm is topology-driven and the second is data-driven. They both use the FF ordering heuristic, *i.e.*, the same as my implementation.

In 2019, a parallel graph coloring algorithm for GPUs that uses a data-centric (Gunrock) abstraction and a linear-algebra-based (GraphBLAS) abstraction were proposed (Osama et al., 2019). They investigated how different optimizations such as hashing, avoiding atomics, and a max-min independent sets impact on the coloring quality and runtime. Gunrock's independent set implementation shows better performance than Naumov *et al.*'s implementation and other optimizations investigated in the paper. A shortcut approach was proposed in 2020 to break data dependencies and thus increase the parallelism of graph coloring (Alabandi et al., 2020). It is almost three times faster on average and uses as few or fewer colors as the best prior GPU codes. However, it is hard to take advantage of this approach to reduce I/O communication.

5. EXPERIMENTAL METHODOLOGY

This thesis compares my code to the fastest serial CPU codes I could find, which are listed in Table 1. As there is no preexisting out-of-core graph coloring code, these codes are the leading in-core graph coloring codes. I measure the runtime of the codes, excluding the time it takes to read the input file from disk and to verify the result. Each experiment is run three times and the median measured runtime is presented.

Table 5.1: The codes that are evaluated

Name	**Version**	**Source**	**Order**
Out-of-Core GC (my code)	1.0	my code	First fit
FirstFit	1.0	(Chen et al., 2017)	First fit
JP-D1		(ColPack, 2019)	Largest degree first
Boost	1.66.0	(Boost, 2009)	First fit

The system I used has dual 10-core 3.1 GHz Xeon E5-2687W v3 CPUs. Each core has separate 32 kB L1 caches, a 256 kB L2 cache, and the cores on a socket share a 25 MB L3 cache. The 128 GB main memory has a peak bandwidth of 68 GB/s. The operating system is Fedora 23. The codes were compiled with gcc/g++ 8.3.1 using "-O3 -march=native".

The codes are evaluated on the 18 graphs listed in Table 2, the rightmost column shows the number of colors used by FF ordering. They are obtained from Center for Discrete Mathematics and Theoretical Computer Science at the University of Rome (Dimacs) (DIMACS, 2010), the Galois framework (Galois) (Galois, 2018), the Stanford Network Analysis Platform (SNAP) (SNAP, 2014), and the SuiteSparse Matrix Collection (SMC)

(SMC, 2011). The graphs are in CSR format. Where necessary, I made the graphs undirected and removed any self-edges.

Table 5.2: Information of the input graphs

Graph name	**Type**	**Origin**	**Vertices**	**Edges**	d_{avg}	d_{max}	**Number of colors**
2d-2e20.sym	grid	Galois	1,048,576	4,190,208	4.0	4	5
amazon0601	co-purchases	SNAP	403,394	4,886,816	12.1	2,752	13
as-skitter	Internet topo.	SNAP	1,696,415	22,190,596	13.1	35,455	76
cit-Patents	patent cites	SMC	3,774,768	33,037,894	8.8	793	15
citationCiteseer	publication	SMC	268,495	2,313,294	8.6	1,318	17
coPapersDBLP	publication	SMC	540,486	30,491,458	56.4	3,299	337
delaunay_n24	triangulation	SMC	16,777,216	100,663,202	6.0	26	9
europe_osm	road map	SMC	50,912,018	108,109,320	2.1	13	5
in-2004	web links	SMC	1,382,908	27,182,946	19.7	21,869	490
internet	Internet topo.	SMC	124,651	387,240	3.1	151	9
kron_g500-logn21	Kronecker	SMC	2,097,152	182,081,864	86.8	213,904	602
r4-2e23.sym	random	Galois	8,388,608	67,108,846	8.0	26	8
rmat16.sym	RMAT	Galois	65,536	967,866	14.8	569	31
rmat22.sym	RMAT	Galois	4,194,304	65,660,814	15.7	3,687	65
soc-LiveJournal1	community	SNAP	4,847,571	85,702,474	17.7	20,333	325
uk-2002	web links	SMC	18,520,486	523,574,516	28.3	194,955	944
USA-road-d.NY	road map	Dimacs	264,346	730,100	2.8	8	5
USA-road-d.USA	road map	Dimacs	23,947,347	57,708,624	2.4	9	5

While it may or may not be useful to color these graphs, I chose them to be able to measure the performance and coloring quality of the codes on a wide variety of graphs. In particular, the number of vertices differs by up to a factor of 776, the number of edges by up to a factor of 1352, the average degree by up to a factor of 41, and the maximum degree by up to a factor of 53,476.

6. RESULTS

This chapter compares the performance of my out-of-core code to the in-core graph coloring codes FirstFit by Chen *et al.* (Chen et al., 2017), ColPack's Jones-Plassmann code with the fastest heuristic (D1) (ColPack, 2019), and the graph coloring code in the Boost library (Boost, 2009). The maximum memory size for my implementation is set to 64 MB in the tests because it is large enough to process some graphs without partitioning and it is small enough to partition most graphs to several chunks. The largest graph is partitioned into more than 100 chunks.

6.1 Coloring Quality

Figure 6.1 shows the number of colors that are used by the 4 different graph coloring codes. The x-axis lists the name of the input graph and the y-axis the number of colors. The smaller the number on the y-axis the better the coloring quality is. The rightmost set of bars reflects the geometric mean over all the inputs.

JP-D1 uses 13% fewer colors than my implementation because JP-D1 implements the Jones-Plassmann algorithm with the largest-degree-first ordering, which tends to be better than FF. My implementation colors the graph with the same number of colors as FirstFit and Boost. This is expected because they are using the same FF ordering heuristic as my code.

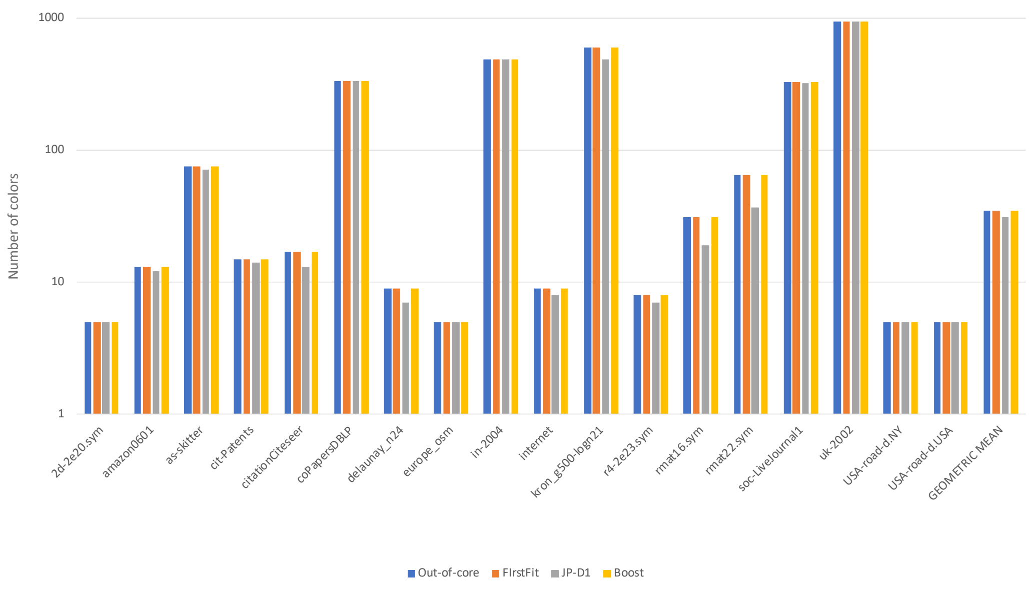


Figure 6.1: Number of colors used by the graph coloring codes

6.2 First and Second Color Usage

Usually, graphs can be colored with a relatively small number of colors and many vertices are colored with the first few colors. Table 3 shows the percentage of vertices ending up with the first or second color. The last row is the geometric mean over all the inputs. It indicates that approximately 40% of the vertices use the first color and over 60% of the vertices use the first two colors. These results validate the use of bit vectors for the first few colors as the majority of the vertices typically end up with one of these colors.

6.3 Throughput

Figure 6.2 presents the throughput of the 4 codes on the Xeon system. The x-axis again lists the name of the input graph, and the y-axis lists the throughput in millions of vertices per second. Higher throughputs are better. The rightmost set of bars presents the geometric mean over all the inputs.

Table 6.1: Percentage of vertices receiving the first two colors

Graph name	**First color**	**First two colors**
2d-2e20.sym	36.5%	73.1%
amazon0601	21.2%	38.6%
as-skitter	49.2%	79.5%
cit-Patents	52.0%	68.6%
citationCiteseer	44.2%	68.1%
coPapersDBLP	10.6%	18.5%
delaunay_n24	24.9%	48.8%
europe_osm	48.1%	95.9%
in-2004	56.1%	80.1%
internet	51.5%	86.2%
kron_g500-logn21	73.8%	83.5%
r4-2e23.sym	27.9%	53.1%
rmat16.sym	36.7%	58.7%
rmat22.sym	42.0%	63.9%
soc-LiveJournal1	43.7%	65.3%
uk-2002	57.5%	77.6%
USA-road-d.NY	43.9%	85.4%
USA-road-d.USA	44.4%	84.0%
GEOMETRIC MEAN	39.3%	64.6%

FirstFit is 4 times faster and JP-D1 is 25% faster than my implementation. This is reasonable considering that they are both in-core codes. They incur no extra workload for partitioning the graph, dealing with local subgraphs, "compressing" color information, searching complex data structures, and copying local results to the global result. Surprisingly, Boost is 6 times slower than my implementation. This indicates that the out-of-core overhead of my implementation is low.

6.4 Throughput for Different Values of R

My implementation has four global parameters: R – the resize function is called when the ratio of the information size to the maximum information size exceeds the value of R,

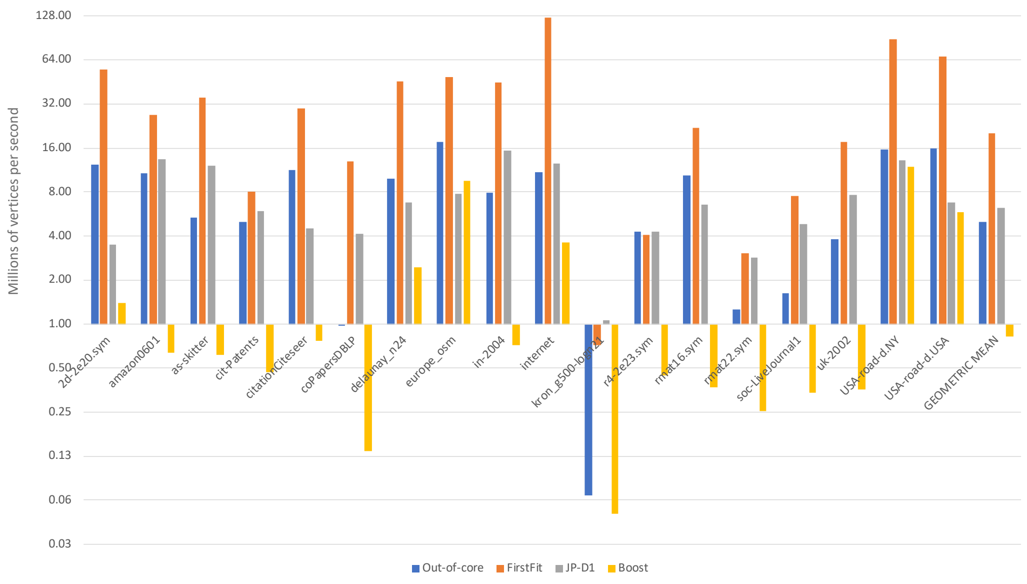


Figure 6.2: Throughput (millions of vertices per second) of the graph coloring codes

C – the number of the bit vectors for all vertices for one color, G – the maximum graph size to total maximum size, and N – the number of bits in the bit vectors. As these global parameters relate to how to assign the available memory and when to resize the information data structure, it is important to evaluate the performance of different values of these parameters. The default values are $R = 0.5$, $C = 2$, $G = 0.5$, and $N = 32 * 16$ as described in Chapter 3.

Figure 6.3 presents the throughput of different values of R. $R = 0.1$ means the information data structure is resized when the ratio of current information size to maximum information size exceeds 0.1. Hence, smaller R values result in more frequent resizing. I evaluated R values between 0.1 and 0.9. In practice, the value of R should not exceed 0.5 to ensure that there is enough memory available for the next iteration.

Most of the 18 graphs exhibit no significant difference in throughput with different values of R. The performance of ”2d-2e20.sym”, ”citationCiteseer”, ”internet”, ”rmat16.sym”,

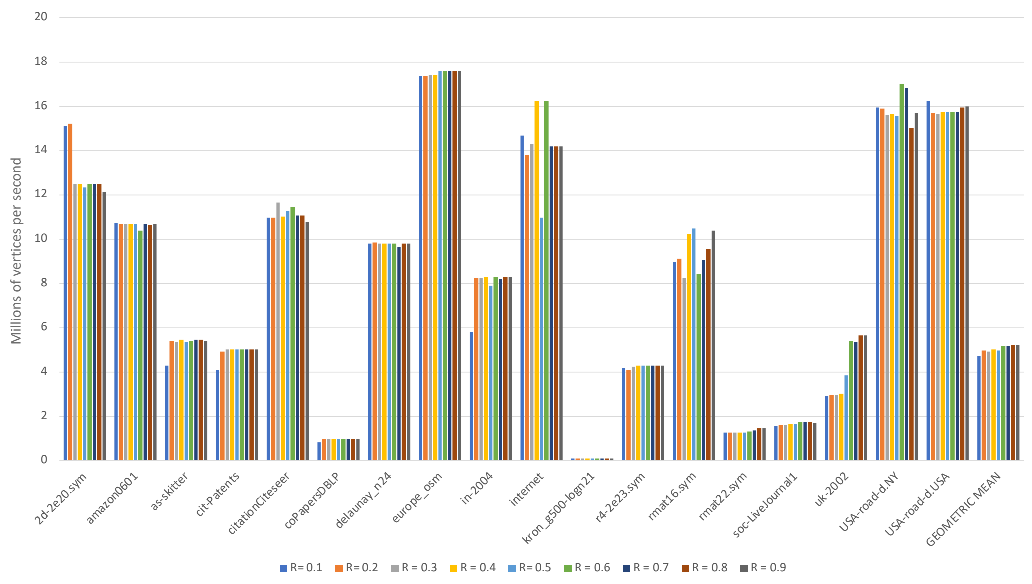


Figure 6.3: Throughput (millions of vertices per second) for different values of R

”uk-2002”, and ”USA-road-d.NY” is more related to the value of R compared to other graphs. They are small graphs, except ”uk-2002”, with a small average degree. The graph ”uk-2002” is the largest graph within the 18 graphs and its performance for $R > 0.5$ is better than the performance for $R < 0.5$. It is because it has a large information data structure and the number of chunks is also significantly greater than for other inputs.

The geometric mean bars show that the throughput grows with the value of R, though not significantly. Given that larger R values cause fewer resize operations, this indicates that resizing is not a performance bottleneck. Considering the performance and the limit on memory, $R = 0.5$ should be a good default value for most graphs.

6.5 Throughput for Different Values of C

The global parameter C is the number of bit vector used for the first few colors. $C = 1$ means there is only one bit-vector to represent the vertices that have the first color.

Figure 6.4 presents the throughput for different values of C. Similar to the measurement of R, the value of C has no significant impact on the performance. The performance of smaller graphs such as ”2d-2e20.sym” and ”internet” is more related to the value of C. The geometric mean shows that the throughput slightly grows from $C = 0$ to $C = 2$ and then stays relatively stable. Hence, $C = 2$ seems to be a good default value.

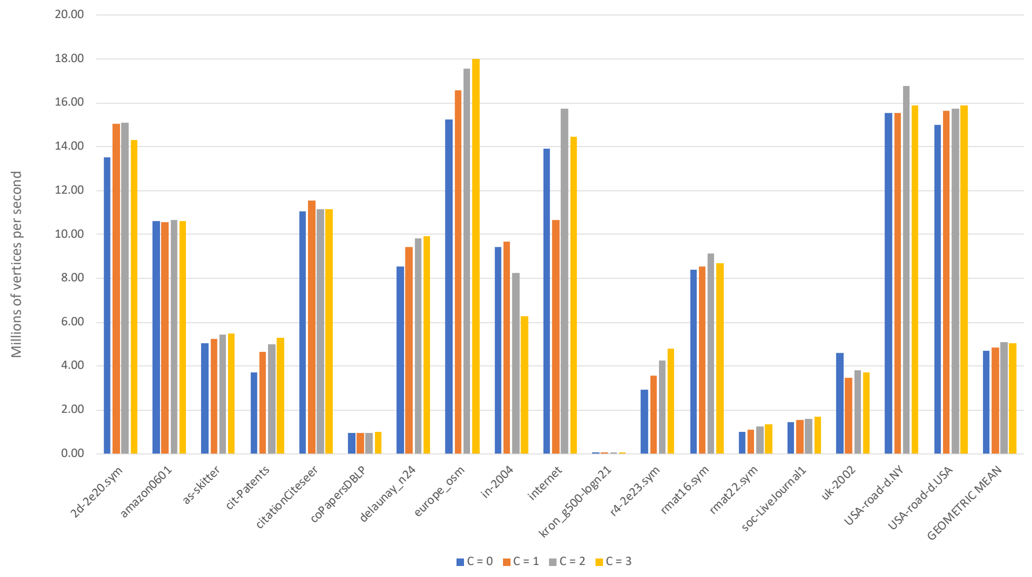


Figure 6.4: Throughput (millions of vertices per second) on different values of C

6.6 Memory Usage for Different Values of C

Considering that memory is critical to the out-of-core graph coloring algorithm and a different number of the bit vectors yields different memory size, the memory usage (in bytes) is also measured.

Figure 6.5 presents the geometric mean of the information size in bytes over each iteration with different values of C. This figure has fewer graphs than the previous one because there are 6 graphs fit in the available memory, which is 64 MB. The size of the information data structure is measured in each iteration. The y-axis is the geometric mean of this information size. The rightmost set of bars is the geometric mean of all the inputs.

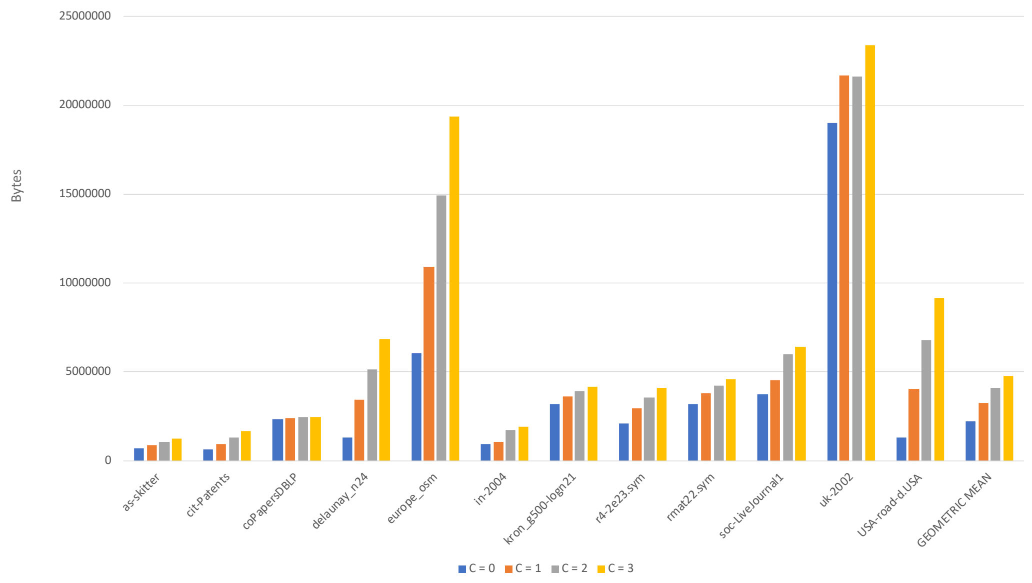


Figure 6.5: Information size (in bytes) for different values of C

The information size increases with increasing values of C for every graph. It indicates that the use of the 3D array and the array of the bit vector is more compact than a simple bit vector for the whole graph. For memory concern, $C = 0$ saves most memory. However, the previous throughput measurement indicates that $C = 2$ has better performance than $C = 1$ and $C = 0$. Hence, C represents a performance and memory trade-off. The best choice depends on the available memory and graph size. For a very large graph with a limited available maximum size, the value of C should be set to 0 to save memory.

6.7 Throughput for Different Values of G

G is the ratio of maximum available size to be assigned to the local subgraph. $G = 0.4$ means 40% of the maximum available memory size is reserved for the local subgraph, *i.e.*, its portion of the CSR arrays. I evaluated values between $G = 0.3$ and $G = 0.9$ as even smaller values of G might not assign enough memory for the local subgraph.

Figure 6.6 shows that most graphs have a quite similar throughput for different values of G. The geometric mean is also quite even. Only the largest graph "uk-2002" is significantly faster when $G = 0.3$. In contrast, "coPapersDBLP" is significantly faster when $G = 0.9$.

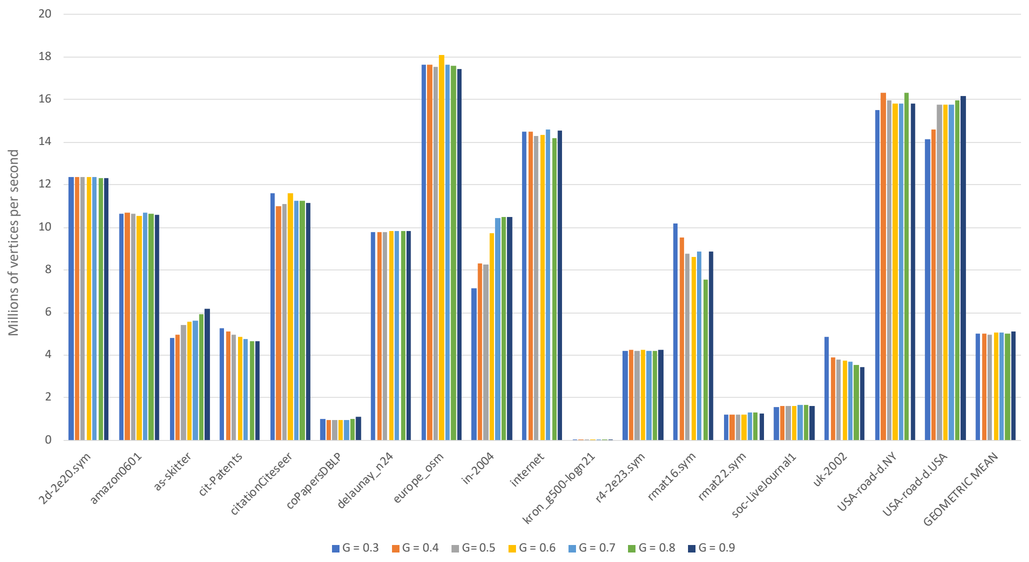


Figure 6.6: Throughput (millions of vertices per second) for different values of G

6.8 Throughput for Different Values of N

N is the number of bits in the bit vectors. As described in Chapters 3.2 and 3.3, the size of the dynamic array is N times smaller than the number of vertices in the associated subgraph. Similarly, the number of chunks in the vector also shrinks for larger values of N. The default value is $N = 32 * 16$. I evaluate $N = 16 * 16$, $N = 32 * 16$, $N = 32 * 32$, and then enlarge 4 times of N until $N = 4096 * 4096$. As presented in Figure 6.7, the performance increases slightly for larger values of N and it decreases when the value of N becomes too large (*e.g.*, $N = 512 * 512$).

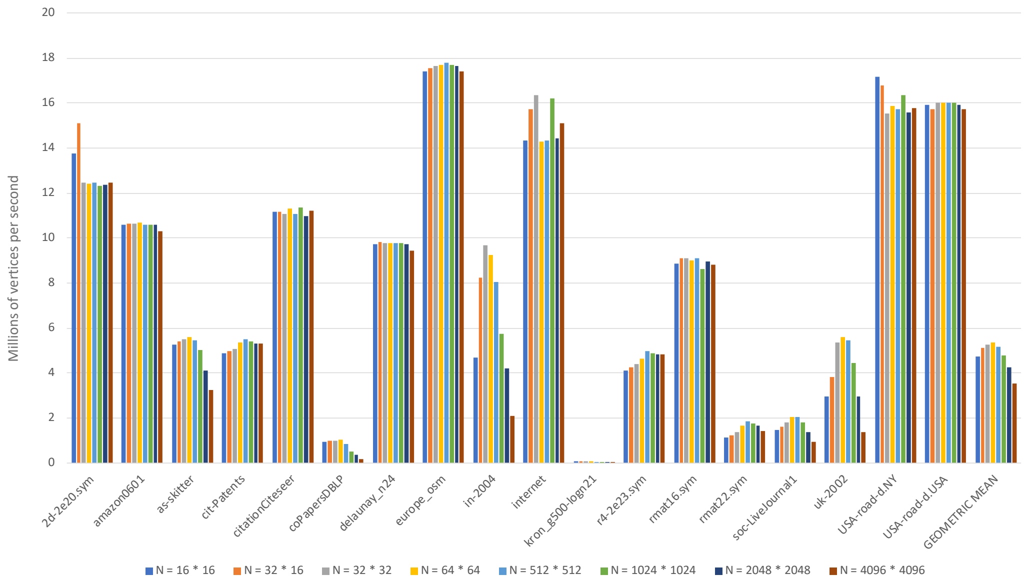


Figure 6.7: Throughput (millions of vertices per second) for different values of N

6.9 Memory Usage for Different Values of N

As discussed above, N impacts the size of both the dynamic array and the vector of chunks. Hence, the information size across each iteration varies for different values of N. The geometric mean of the information size across each iteration is measured and presented in Figure 6.8. The rightmost column of bars is the geometric mean over all the inputs. The y-axis is logarithmic. As expected, the figure shows that the information size decreases for larger N. As the memory is limited, a smaller information size yields a larger subgraph. The memory size decreases until $N = 512 * 512$. This indicates that large values of N are preferable to save memory. However, the value of N can not be too large for efficiency reasons.

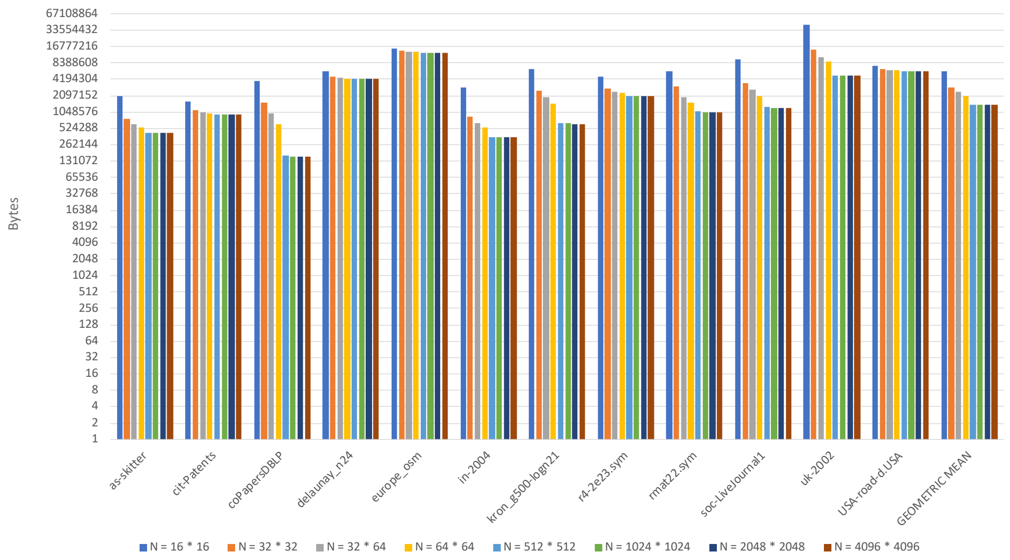


Figure 6.8: Information size (in bytes) for different value of N

6.10 Resize Frequency

I profiled the code and found that the resize function is the most expensive function. Hence, the call frequency of the resize function is measured. Figure 6.9 only shows results for the 12 graphs that are partitioned (the remaining 6 graphs are small enough to fit in the allocated memory). Most graphs do not ever have to resize the vector during processing. For the 4 graphs that do call the resize function, the frequency of resizing is usually about half of the number of subgraphs. These results show that the expensive resizing function can often be avoided to improve performance.

6.11 Memory Usage

The memory usage for 3 of the input graphs is evaluated. The graph ”kron_g500-logn21.egr” is evaluated because it is a large graph and has the lowest throughput of all tested inputs. The graphs ”r4-2e23.sym.egr” and ”rmat22.sym.egr” are evaluated

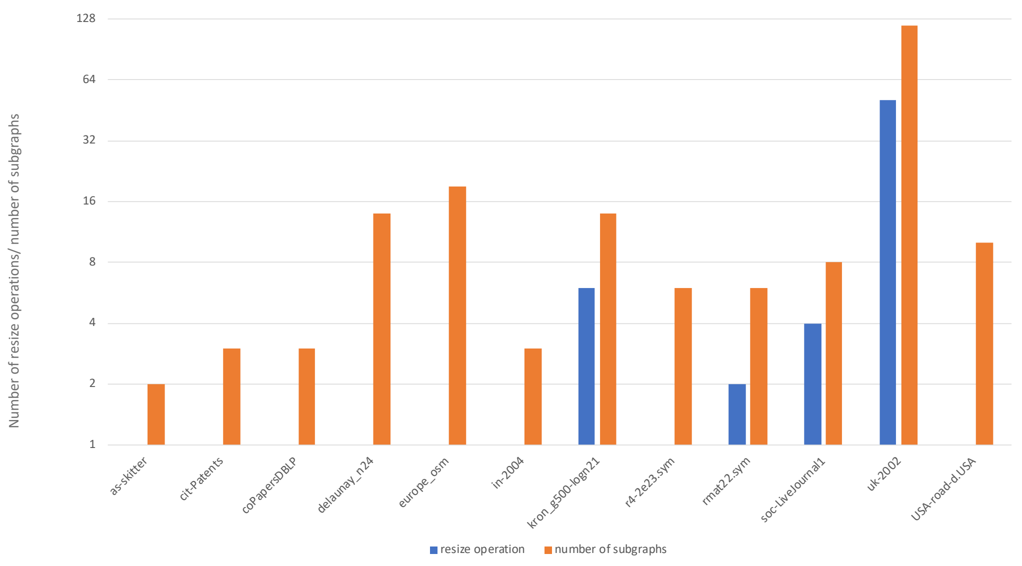


Figure 6.9: Frequency to call the resize function

because they are partitioned into the same number of subgraphs (6 subgraphs) but call the resize function different number of times, *i.e.*, ”r4-2e23.sym.egr” is never resized but ”rmat22.sym.egr” calls resize twice (after processing subgraph 3 and subgraph 4). The graph size, maximum graph size, information size, and maximum information size for each subgraph (iteration) are presented in Figure 6.10, Figure 6.11, and Figure 6.12. The information size is calculated after processing the subgraph and graph size is determined in the graph partitioning before processing the subgraph. For example, the information size of subgraph 2 is the size of the information after processing subgraph 2.

Even though these three graphs are very different in graph size, throughput, and frequency of resizing, their memory usage trends are quite similar. The maximum graph size is dynamic to leave enough memory to store information and import as large a subgraph as possible. For example, Figure 6.10 shows that the information size is quite small in the first iteration. The maximum graph size is therefore expanded in the next iteration to import a larger subgraph. The maximum graph size is becoming smaller in later iterations as the information size grows. Also, the information size usually does not exceed half of the maximum information size.

The maximum graph size increases after processing subgraph 1 in both Figure 6.11 and Figure 6.12. However, the maximum graph size remains the same after subgraph 1 in Figure 6.11 and the maximum graph size decreases in later iterations in Figure 6.12.

The ratio of information size to maximum information size exceeds 0.5 in the subgraph as presented in Figure 6.12. Hence, it calls the resize function after processing subgraph 3. This ratio still exceeds 0.5 after the resizing and the maximum graph size is, therefore, decreased in the next iteration which is subgraph 4. Similarly, this ratio exceeds 0.5 again in subgraph 4 and the resize is called. The maximum graph size decreases in subgraph 5 as this ratio is still relatively large (*e.g.*, exceeds the value of R, which is 0.5). As long as the information size is smaller than the maximum information size in subgraph 5, the maximum graph size of subgraph 6 will not decrease because the information size will not increase in the last subgraph. As there is a small part of the graph left, the graph size of the last subgraph in Figure 6.12 is relatively small compared to other subgraphs.

This similar memory usage trends indicate that the partitioning strategy and resize function work as expected to maintain the ratio of information size to maximum information size within the value of R, which is 0.5 by default.

6.12 Comparing to store the color in an array

The simplest way to record the vertex ID with its associate color is to record the color in an array. The size of the array is the number of vertices in the whole graph so that the color can be accessed directly with the vertex ID. Hence, I am curious about if my

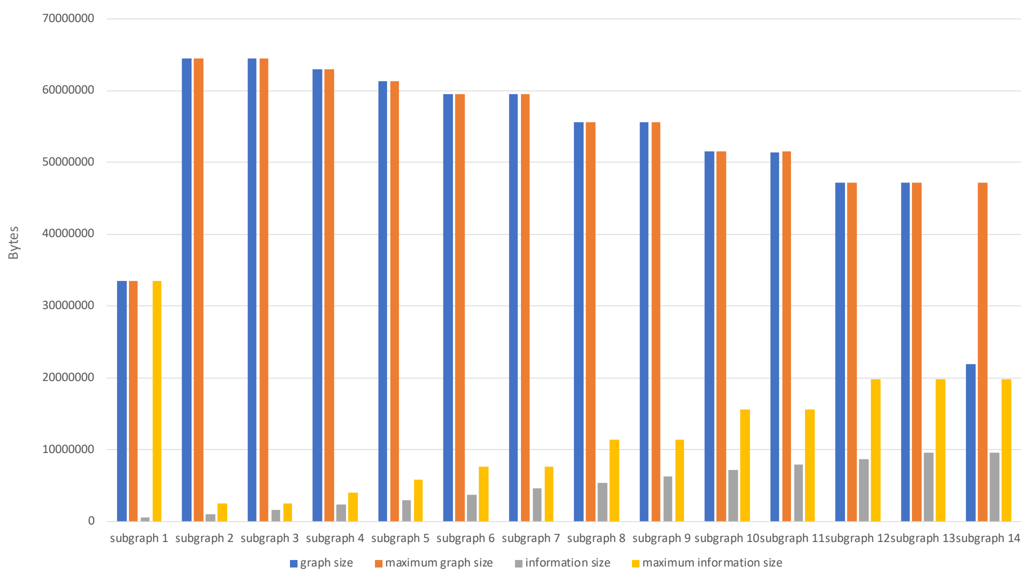


Figure 6.10: Memory usage (in bytes) for graph ”kron_g500-logn21.egr” across each iteration

implementation has a smaller size comparing to store the color in an array. The y-axis in Figure 6.13 is the ratio of my size of implementation to the size of storing the color in an array. The orange columns present the 3 graphs that my implementation is worse than a color array. These 3 graphs are all having relatively large average degrees comparing to their graph size. Hence, my implementation saves more memory for most graphs but might hurt with the graphs that have relatively large average degrees.

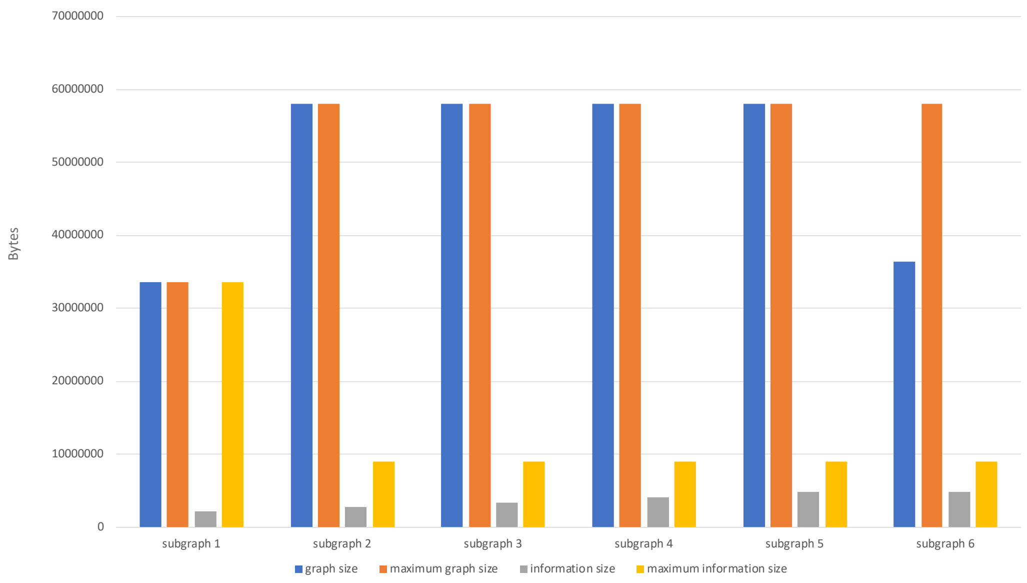


Figure 6.11: Memory usage (in bytes) for graph ”r4-2e23.sym.egr” across each iteration

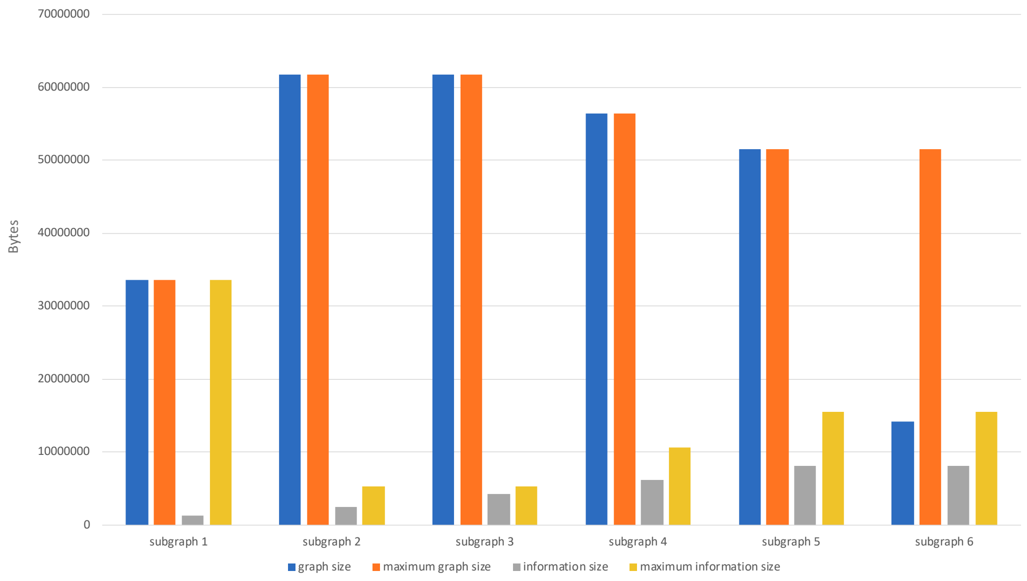


Figure 6.12: Memory usage (in bytes) for graph ”rmat22.sym.egr” across each iteration

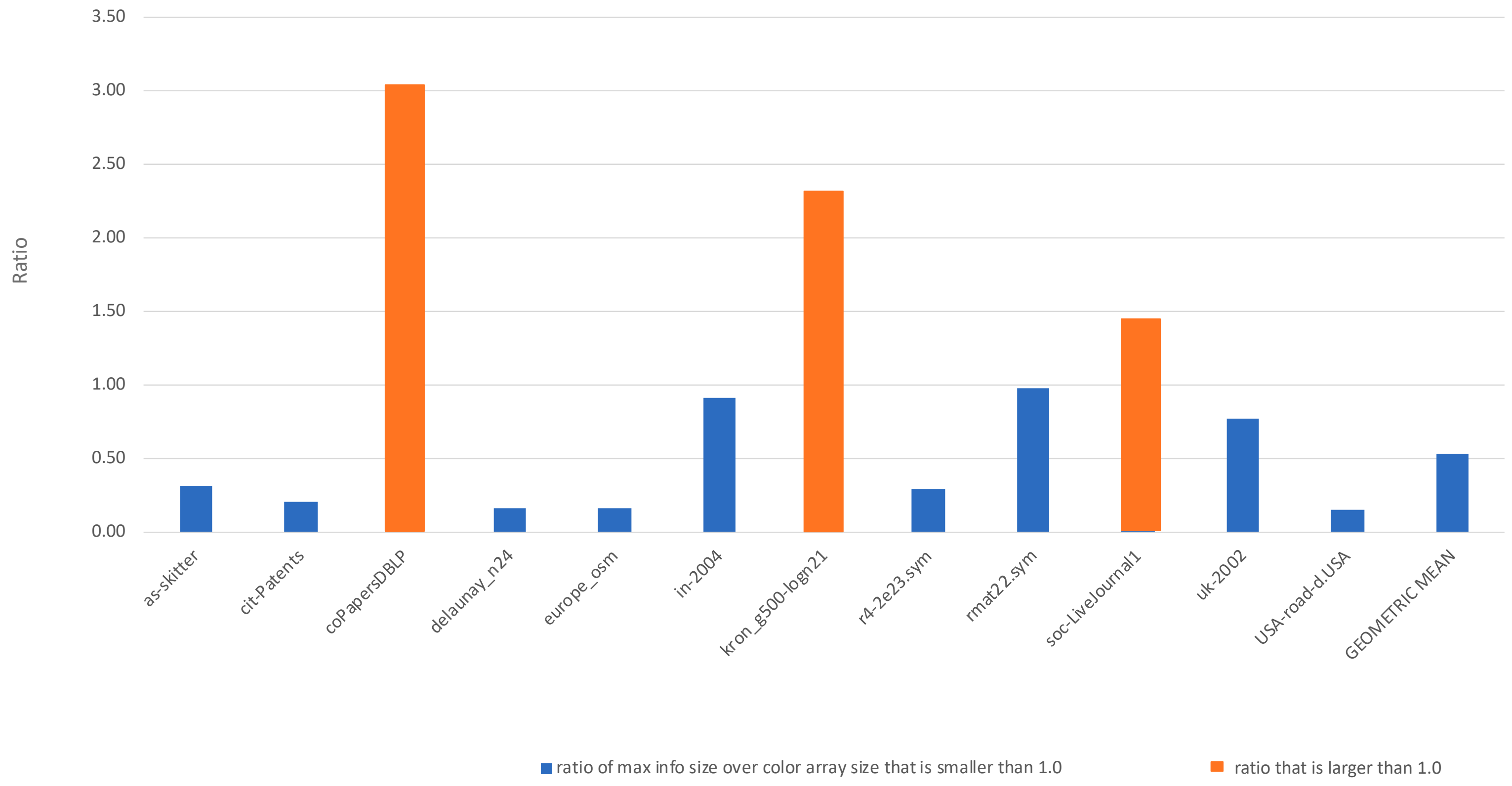


Figure 6.13: Ratio of the maximum information size over simply storing the colors in an array

7. SUMMARY AND CONCLUSIONS

With the constant growth of data sets, graphs such as social networks become larger. Especially Internet-related graphos are often too massive to fit in the memory of a single computer. This thesis proposes an out-of-core graph coloring algorithm. It dynamically partitions the graph so that each subgraph fits in the available memory and leaves part of the memory to record information for later subgraphs. This algorithm guarantees that the result is the same as processing the graph as a whole.

To save memory and boost performance, I designed two data structures to compactly record the color of each vertex that has neighbors in later subgraphs. The basic idea is that each color has an associated set of vertices that use this color and have neighbors in later subgraphs. As the number of colors used for the graph is usually small, the number of sets is also small, which is good for searching. For the first a few colors, every color employs a chunked bit-vector. A set bit indicates that the corresponding vertex has this color. For the later, less frequently used colors, the compressed lists is used to represent the vertices. The dynamic array stores the index of the shared vector. I efficiently resize the vector to minimize its size.

Three global parameters are introduced to determine how to partition the graph, when to resize the array, and how many colors should use the bit vectors to record their associated vertices. A study of these three parameters indicates that different values have no significant relation to the resulting performance. However, different graph sizes do tend to prefer specific settings of these parameters.

The value of N, which is the number of bit in the bit vector for each chunk impacts the information size and the performance. Usually, it saves more memory and yields better performance with a larger N (*e.g.*, $N = 64 * 64$ is better than $N = 32 * 32$), but a very large N would hurt the performance (*e.g.*, $N = 2048 * 2048$).

Compared to 3 leading in-core graph coloring algorithms running on a Xeon system, JP-D1 is 25% faster and uses 13% fewer colors than my out-of-core algorithm. FirstFit and Boost both use the same number of colors as my implementation, but FirstFit is 4 times faster whereas Boost is 6 times slower. Hence, I believe the out-of-core overhead is reasonable.

In conclusion, this work enables the processing of massive graphs given a limited memory resource. A similar approach may be useful for implementing other graph algorithms such as maximal independent set (MIS) computation. Also, another avenue for future work would be to delete the dynamic arrays that are not used to save memory and parallelize my implementation for better performance.

BIBLIOGRAPHY

Alabandi, G., Powers, E., and Burtscher, M. (2020). Increasing the parallelism of graph coloring via shortcutting. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 262–275.

Boost (2009). `https://www.boost.org/doc/libs/1_63_0/libs/graph_parallel/doc/html/index.html`. Accessed: 6/25/2020.

Che, S., Rodgers, G., Beckmann, B., and Reinhardt, S. (2015). Graph coloring on the gpu and some techniques to improve load imbalance. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 610–617. IEEE.

Chen, X., Li, P., Fang, J., Tang, T., Wang, Z., and Yang, C. (2017). Efficient and high-quality sparse graph coloring on gpus. *Concurrency and Computation: Practice and Experience*, 29(10):e4064.

ColPack (2019). Combinatorial scientific computing and petascale simulations. `https://github.com/CSCsw/ColPack`. Accessed: 6/25/2020.

DIMACS (2010). Center for discrete mathematics and theoretical computer science. `http://www.dis.uniroma1.it/challenge9/download.shtml`. Accessed: 4/30/2020.

Galois (2018). The university of texas at austin. `https://iss.oden.utexas.edu/?p=projects/galois`. Accessed: 4/30/2020.

Gebremedhin, A. H., Nguyen, D., Patwary, M. M. A., and Pothen, A. (2013). Colpack: Software for graph coloring and related problems in scientific computing. *ACM Transactions on Mathematical Software (TOMS)*, 40(1):1–31.

Gharaibeh, A., Reza, T., Santos-Neto, E., Costa, L. B., Sallinen, S., and Ripeanu, M. (2013). Efficient large-scale graph processing on hybrid cpu and gpu systems. *arXiv preprint arXiv:1312.3018*.

Grosset, A. V. P., Zhu, P., Liu, S., Venkatasubramanian, S., and Hall, M. (2011). Evaluating graph coloring on gpus. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, pages 297–298.

Han, W., Mawhirter, D., Wu, B., and Buland, M. (2017). Graphie: Large-scale asynchronous graph traversals on just a gpu. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 233–245. IEEE.

Hasenplaugh, W., Kaler, T., Schardl, T. B., and Leiserson, C. E. (2014). Ordering heuristics for parallel graph coloring. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*, pages 166–177.

Iwabuchi, K., Sato, H., Mizote, R., Yasui, Y., Fujisawa, K., and Matsuoka, S. (2014). Hybrid bfs approach using semi-external memory. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, pages 1698–1707. IEEE.

Jones, M. T. and Plassmann, P. E. (1993). A parallel graph coloring heuristic. *SIAM Journal on Scientific Computing*, 14(3):654–669.

Kim, M.-S., An, K., Park, H., Seo, H., and Kim, J. (2016). Gts: A fast and scalable graph processing method based on streaming topology to gpus. In *Proceedings of the 2016 International Conference on Management of Data*, pages 447–461.

Kyrola, A., Blelloch, G., and Guestrin, C. (2012). Graphchi: Large-scale graph computation on just a {PC}. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 31–46.

Lewis, R. (2015). *A guide to graph colouring*, volume 7. Springer.

Lin, J., Cai, S., Luo, C., and Su, K. (2017). A reduction based method for coloring very large graphs. In *IJCAI*, pages 517–523.

Luby, M. (1986). A simple parallel algorithm for the maximal independent set problem. *SIAM journal on computing*, 15(4):1036–1053.

Ma, L., Yang, Z., Chen, H., Xue, J., and Dai, Y. (2017). Garaph: Efficient gpu-accelerated graph processing on a single machine with balanced replication. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 195–207, Santa Clara, CA. USENIX Association.

Maass, S., Min, C., Kashyap, S., Kang, W., Kumar, M., and Kim, T. (2017). Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 527–543.

Merrill, D., Garland, M., and Grimshaw, A. (2012). Scalable gpu graph traversal. *Acm Sigplan Notices*, 47(8):117–128.

Naumov, M., Castonguay, P., and Cohen, J. (2015). Parallel graph coloring with applications to the incomplete-lu factorization on the gpu. *Nvidia White Paper*.

Osama, M., Truong, M., Yang, C., Buluç, A., and Owens, J. (2019). Graph coloring on the gpu. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 231–240. IEEE.

SMC (2011). Suitesparse matrix collection. `https://sparse.tamu.edu/`. Accessed: 4/30/2020.

SNAP (2014). Stanford large network dataset collection. `https://snap.stanford.edu/data/`. Accessed: 4/30/2020.

Vitter, J. S. (2001). External memory algorithms and data structures: Dealing with massive data. *ACM Computing surveys (CsUR)*, 33(2):209–271.

Yoneki, E., Nilakant, K., Dalibard, V., and Roy, A. (2014). Prefedge: Ssd prefetcher for large-scale graph traversal. In *Proceedings of the International Conference on Systems and Storage (SYSTOR'14)*, pages 1–12.

Zuckerman, D. (2006). Linear degree extractors and the inapproximability of max clique and chromatic number. In *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, pages 681–690.