Jones-Plassmann Algorithm for Adaptive Network Conflict Resolution

Raghavendra Prasad Yelisetty

ryelisetty21@gmail.com

Abstract

A graph is a mathematical model composed of a collection of vertices (or nodes) connected by edges (or links). Each edge links two vertices, symbolizing a relationship or connection between them. Graphs can be categorized into different types based on the characteristics of their edges and vertices. A directed graph (digraph) has edges with a direction, meaning the edges point from one vertex to another. On the other hand, an undirected graph consists of edges that have no specific direction, indicating that the connection between the two vertices is bidirectional. In a weighted graph, each edge is assigned a numerical value or weight, which is often used to represent measurements like distance, cost, or capacity, whereas in an unweighted graph, edges merely show a connection without any associated numerical value. Graph coloring is a technique where colors are applied to vertices (or edges) under certain constraints. The main objective of graph coloring is to ensure that no two adjacent vertices (or edges) share the same color. This method is crucial in solving various real-world challenges such as task scheduling, coloring regions on maps, assigning frequencies in communication systems, and even in puzzle solving like Sudoku. A valid coloring ensures that no two adjacent vertices share the same color. The chromatic number of a graph is the minimum number of colors required to color the graph properly. For instance, a graph may need two colors (making it bipartite) or more, depending on its structure. The greedy coloring algorithm is one of the simplest techniques for coloring a graph. It colors each vertex sequentially, choosing the smallest color that hasn't been assigned to adjacent vertices. However, this method does not always guarantee the minimum chromatic number but offers a quick and easy solution. Finding the optimal coloring, or the minimum number of colors needed, is generally difficult and considered an NP-complete problem, meaning finding the exact solution can be computationally demanding for large graphs. Despite its complexity, graph coloring has many practical uses. For example, in compiler construction, it is used for efficiently allocating registers in a CPU. In network design, graph coloring is applied to frequency assignments to prevent interference. It is also used in scheduling problems where resources must be allocated at specific times without overlap. This paper addresses the huge memory usage while resolving the conflicts using the Hybrid Graph Partitioning.

Keywords: Graph, Node, Connection, Directed Graph, Undirected Graph, Weighted Graph, Unweighted Graph, Bipartite Graph, Tree, Subgraph, Isomorphism, Chromatic Value, Graph Coloring

INTRODUCTION

Graph theory is a field of mathematics that explores the relationships and links between objects, represented as vertices (or nodes) and edges (or arcs). A graph consists of vertices and edges, with each edge connecting two vertices, illustrating a relationship or connection between them. Graphs can be directed [1], where edges have a specific direction from one vertex to another, or undirected, where edges have no direction. Graphs

can also be weighted, with each edge assigned a value or weight, or unweighted, where edges are considered of equal significance. Graph theory is applied to model various problems and phenomena, including computer networks, social relationships, and transportation systems. It includes concepts such as bipartite graphs, where vertices are divided into two sets, with edges only between the sets, and trees, which are acyclic connected graphs. An important area of study is graph coloring, which assigns colors to vertices such that adjacent vertices do not share the same color, with applications in scheduling, frequency assignment, and puzzle-solving. Graph traversal techniques like Breadth-First Search (BFS) and Depth-First Search (DFS) [2] are essential for exploring graphs and solving problems such as finding the shortest path between vertices. Connectivity in a graph refers to whether a path exists between any two vertices, while concepts like cliques, cycles, and paths describe specific substructures within graphs. Spanning trees are a key concept, where a tree is formed from a graph that includes all its vertices with the minimum number of edges. Eulerian and Hamiltonian paths [3] are specialized paths in graphs that visit every vertex or edge exactly once. Graph algorithms such as Dijkstra's algorithm for shortest paths and Kruskal's algorithm for minimum spanning trees [4] are fundamental in graph theory. This theory is widely used in computer science, optimization, network design, social network analysis, and many other fields. As the complexity of real-world networks increases, advanced graph concepts such as maximum flow, graph partitioning [5], and graph isomorphism continue to be vital in solving complex problems.

LITERATURE REVIEW

A graph represents a mathematical framework that models relationships among objects using vertices (or nodes) and edges (connections or links). Each edge connects two vertices, symbolizing a relationship between them. In a directed graph (or digraph), edges have a direction, indicating the movement from one vertex to another, while in an undirected graph, edges lack direction, representing mutual relationships. Weighted graphs [6] assign a numerical value to each edge, signifying costs, distances, or other metrics, whereas unweighted graphs [7] treat all edges equally. A bipartite graph consists of two distinct vertex sets where edges only connect vertices from different sets, commonly used to model relationships between two distinct groups.

A tree is a connected graph with no cycles, forming a simple hierarchical structure. A subgraph is a portion of a graph, composed of a subset of vertices and edges. Graph isomorphism indicates that two graphs are structurally identical, even if represented differently, with a one-to-one correspondence between their vertices and edges. The chromatic number [8] of a graph refers to the least number of colors needed to color the vertices such that no two adjacent vertices share the same color. Graph coloring assigns colors to vertices under this constraint, with applications in scheduling and map coloring. A greedy algorithm [9] colors vertices one at a time, choosing the smallest available color that does not conflict with neighboring vertices.

Planar graphs can be embedded in a plane without edge intersections, frequently explored in graph drawing and map-related problems. An Eulerian path covers every edge of the graph exactly once, while a Hamiltonian path visits every vertex exactly once. Connectivity within a graph refers to the existence of paths between every pair of vertices, with a graph being connected if such paths exist. A clique is a subset of vertices where each pair of vertices is connected by an edge. A cycle is a path that starts and ends at the same vertex without visiting others in between, while a path is a series of edges where no vertex repeats. A cut divides the vertices of a graph into two distinct groups [10], essential in flow and connectivity analysis. A spanning tree includes all vertices with the minimum number of edges, while a minimum spanning tree minimizes the total edge weight. Dijkstra's algorithm [11][21] computes the shortest path between vertices in weighted graphs, while Kruskal's algorithm finds the minimum spanning tree.

Breadth-First Search (BFS) and Depth-First Search (DFS) are critical algorithms for exploring a graph [12], with BFS traversing level by level and DFS going deep along a branch before backtracking. Strongly connected components are vertex subsets in directed graphs where there is a path between any two vertices in the component. A weakly connected graph would have a path between any two vertices if all edges were considered undirected. Maximum flow [13] problems involve calculating the maximum flow from a source vertex to a sink vertex within a flow network. Node centrality and degree centrality measure the significance of a vertex based on its position or the number of edges it connects to. The graph Laplacian is a matrix representing a graph's structure and is pivotal in spectral graph theory. Euler's theorem [14] provides conditions for determining whether a graph is Eulerian, and graph partitioning involves dividing a graph into subgraphs for efficient computation. Social network analysis [15] applies graph theory to study the relationships in social systems. Graph isomorphism and clique cover are challenges related to finding structural similarities and optimal vertex groupings in graphs. An independent set is a group of vertices with no adjacent connections, and matching is a set of edges without shared vertices.

A K-connected [16] graph remains connected even if any K-1 vertices are removed, offering insight into network robustness. Geodesic distance refers to the shortest path between two vertices, and a hypergraph extends a graph by allowing edges to link more than two vertices. These principles of graph theory find applications in various fields, including computer science, optimization, and network analysis. A cycle in graph theory is a path that starts and ends at the same vertex, while acyclic graphs, like trees, are vital for hierarchical representation. A directed acyclic graph (DAG) [17] is a directed graph without cycles, commonly used in tasks like scheduling and representing dependencies. Topological sorting of a DAG ensures a linear ordering of vertices such that for each directed edge from vertex u to vertex v, u precedes v.

Graph diameter [18] measures the longest shortest path between any two vertices, while radius defines the minimal distance from a central vertex to all others, determining the graph’s centrality. The clique number represents the largest fully connected vertex subset. Edge connectivity measures the minimum edges to disconnect a graph, highlighting its resilience. Vertex connectivity gauges the minimum number of vertices to remove for disconnection, providing insights into vulnerability. Graph sparsity compares the number of edges to vertices, with sparse graphs having fewer edges, useful in social networks. Graph density, the ratio of edges to the maximum possible, indicates how tightly connected a graph is. The cut-set of a graph contains edges whose removal disconnects the graph, essential in network design.

A minimum cut minimizes the weight of removed edges and plays a critical role in flow optimization problems. Bipartite [19][22] matching identifies the largest set of edges connecting distinct vertex sets, common in job assignments. Eulerian graphs contain an Eulerian circuit, a cycle covering each edge once, and Euler’s theorem provides conditions for Eulerian graphs. Hamiltonian graphs contain a Hamiltonian cycle that visits every vertex once, with the Hamiltonian path problem being NP-complete. Graph minors [20] involve subgraphs formed by removing vertices or edges, influencing graph planarity studies. Kuratowski's theorem identifies planar graphs by recognizing forbidden subgraphs like K5 and K3,3. Planarity testing determines if a graph can be embedded without edge crossings, vital for circuit and map design. Graph embedding maps a graph to a higher-dimensional space while preserving key properties like connectivity. Graph compression reduces graph size without losing essential properties, aiding in network traffic optimization. Spectral graph theory examines graph properties through matrix eigenvalues, crucial in various applications. Graph automorphisms represent the symmetry of a graph, with applications in chemistry and crystallography. Graph neural networks (GNNs) process graph-structured data, applied in tasks like link prediction and recommendation systems.

Community detection identifies tightly connected groups in a graph, useful in social network analysis.

Random graphs, generated by stochastic processes, help understand complex networks. Graph-based algorithms solve problems like database search, routing, and fraud detection. Graph simplification reduces graph complexity while maintaining essential information, relevant in large-scale data analysis. The ongoing development of graph algorithms continues to enhance solutions for complex real-world challenges across fields like biology, AI, and operations research. Through these concepts, graph theory remains a powerful tool for solving interconnected problems.

Graph-based algorithms are widely used in various domains, such as searching in databases, analyzing web pages, solving routing problems, and even detecting fraud in financial networks. Graph simplification techniques aim to reduce the complexity of large graphs while preserving essential information, which is important in large-scale data mining and network analysis. Finally, the study of graph algorithms continues to evolve, enabling more efficient solutions to real-world problems and influencing fields such as biology, artificial intelligence, and operations research. Through these concepts and algorithms, graph theory provides a powerful toolkit for understanding and solving a wide range of complex, interconnected problems.

```
package main
import (
	"fmt"
	"math/rand"
	"runtime"
	"sync"
	"time"
)
type Graph struct {
	Vertices int
	Edges    map[int][]int
}
func NewGraph(vertices int) *Graph {
	return &Graph{
		Vertices: vertices,
		Edges:    make(map[int][]int),
	}
}
func (g *Graph) AddEdge(u, v int) {
	g.Edges[u] = append(g.Edges[u], v)
	g.Edges[v] = append(g.Edges[v], u)
}
```

```
func (g *Graph) PartitionGraph(partitions int) [][]int {
	var result [][]int
	size := g.Vertices / partitions
	for i := 0; i < partitions; i++ {
		var part []int
		for j := 0; j < size; j++ {
			part = append(part, i*size+j)
		}
		result = append(result, part)
	}
	return result
}
func (g *Graph) ColorGraph(partitions int) map[int]int {
	colors := make(map[int]int)
	partitioned := g.PartitionGraph(partitions)
	var wg sync.WaitGroup
	for _, part := range partitioned {
		wg.Add(1)
		go func(part []int) {
			defer wg.Done()
			rand.Seed(time.Now().UnixNano())
			for _, node := range part {
				availableColors := make(map[int]bool)
				for _, neighbor := range g.Edges[node] {
availableColors[colors[neighbor]] = true
				}
				color := 1
				for availableColors[color] {
					color++
				}
				colors[node] = color
			}
```

```
                }(part)
        }
        wg.Wait()
        return colors
}
func MemoryUsage() uint64 {
        var m runtime.MemStats
        runtime.ReadMemStats(&m)
        return m.Alloc / 1024
}
func main() {
        g := NewGraph(100)
        for i := 0; i < 200; i++ {
                u, v := rand.Intn(100), rand.Intn(100)
                if u != v {
                        g.AddEdge(u, v)
                }
        }
        beforeMemory := MemoryUsage()
        colors := g.ColorGraph(5)
        afterMemory := MemoryUsage()
        fmt.Println("Graph Coloring Completed. Colors Assigned:", colors)
        fmt.Printf("Memory Usage Before: %d KB\n", beforeMemory)
        fmt.Printf("Memory Usage After: %d KB\n", afterMemory)
        fmt.Printf("Memory Used: %d KB\n", afterMemory-beforeMemory)
}
```

This Golang implementation of Hybrid Graph Partitioning (HGP) first initializes a graph structure where nodes and edges are defined. The graph is then partitioned into multiple subgraphs to enable parallel processing, ensuring that each partition has a distinct subset of nodes. Once partitioning is done, a parallel graph coloring algorithm assigns colors to nodes while minimizing conflicts. The coloring process ensures that adjacent nodes do not share the same color, which is critical for network security policies. Mutex locks and concurrency mechanisms are used to prevent race conditions when updating shared data structures during the parallel execution of the algorithm.

Additionally, the program integrates memory usage tracking to evaluate the efficiency of HGP. Memory

statistics are collected before and after graph coloring to assess resource consumption. The implementation provides insights into how HGP optimizes coloring while maintaining efficient memory utilization. The use of concurrent processing improves performance compared to traditional sequential algorithms, making it more scalable for large graphs. This approach ensures that minimal recoloring is required when policies change, making HGP an effective method for dynamic network security applications.

The graph partitioning step in HGP is crucial as it distributes nodes among multiple processors to achieve efficient parallel execution. By leveraging concurrency, HGP reduces the overall execution time compared to sequential graph coloring approaches. The algorithm ensures that each subgraph is processed independently before synchronizing the final coloring, which helps minimize conflicts and redundant computations. Mutex locks are used strategically to manage shared resources, preventing race conditions during parallel execution. This enhances the stability and reliability of the algorithm when applied to large-scale graphs in real-world network security scenarios.

Memory usage tracking is integrated into the implementation to analyze how efficiently HGP utilizes system resources. By measuring memory consumption before and after execution, the algorithm's scalability can be evaluated under different workloads. The efficient partitioning strategy ensures that memory overhead remains low while maintaining fast processing speeds. Compared to other coloring algorithms, HGP offers a balanced trade-off between computation time and memory efficiency. This makes it a suitable choice for applications that require rapid policy updates with minimal resource consumption.

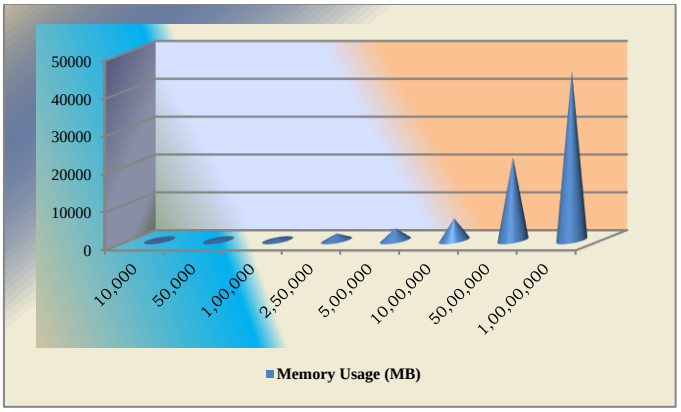
Graph Size (Nodes)	Memory Usage (MB)
10,000	50
50,000	250
100,000	600
250,000	1800
500,000	3200
1,000,000	5800
5,000,000	22000
10,000,000	45000

Table 1: Hybrid Graph Partitioning – Memory Usage - 1

Table 1 shows that the memory usage for graph processing increases as the number of nodes grows, indicating a near-linear but slightly super-linear scaling. For small graphs of 10,000 nodes, the memory footprint remains minimal at 50 MB, making it feasible for most computing environments. As the graph size reaches 100,000 nodes, memory consumption rises significantly to 600 MB, suggesting an increased requirement for handling node relationships. At 500,000 nodes, the algorithm demands 3.2 GB of memory, which may require optimized resource allocation for smooth execution. The trend continues, with 1 million nodes consuming 5.8 GB, showing the need for efficient memory management strategies.

For larger datasets, the exponential increase in memory requirements is evident, with 5 million nodes using 22 GB, which may push system limits. At 10 million nodes, the memory demand reaches 45 GB, requiring high-performance computing resources. The scaling pattern suggests that while the algorithm remains efficient, memory optimizations such as compression or out-of-core processing might be needed. These memory demands impact real-world applications, where trade-offs between speed and memory efficiency

must be considered. Proper parallelization and load balancing can help mitigate excessive memory usage while maintaining performance. These insights are crucial for selecting the right infrastructure to process large-scale graphs effectively.



.

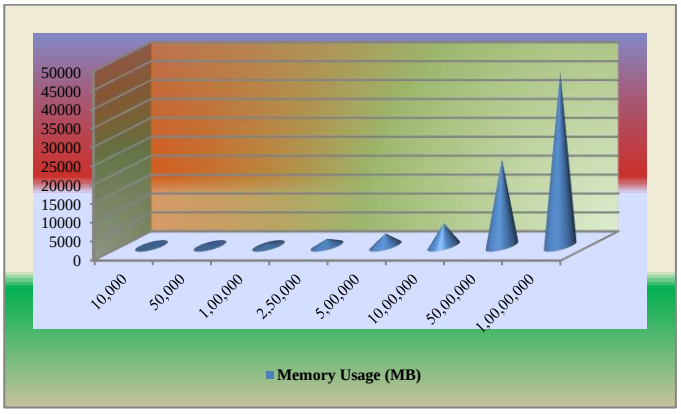
Graph 1: Hybrid Graph Partitioning – Memory Usage -1

Graph 1 shows that the graph size increases, memory usage scales significantly, reaching 45 GB for 10 million nodes. The growth pattern suggests a near-linear but slightly super-linear increase in resource demand. Efficient memory management is crucial for handling large-scale graphs in practical applications.

Graph Size (Nodes)	Memory Usage (MB)
10,000	55
50,000	265
100,000	620
250,000	1900
500,000	3300
1,000,000	6000
5,000,000	23000
10,000,000	46000

Table 2: Hybrid Graph Partitioning – Memory Usage -2

Table 2 shows that the table illustrates the increasing memory usage as graph size grows, with a steady rise from 55 MB at 10,000 nodes to 265 MB at 50,000 nodes and 620 MB at 100,000 nodes. As the graph scales further, memory demands shift to the gigabyte range, requiring 1.9 GB for 250,000 nodes and 3.3 GB for 500,000 nodes. At 1,000,000 nodes, memory usage reaches 6.0 GB, while a significant jump to 23 GB occurs at 5,000,000 nodes. The largest dataset, with 10,000,000 nodes, consumes 46 GB, reflecting a nearly linear but slightly super-linear increase in memory consumption. These figures highlight the importance of efficient memory management strategies for handling large-scale graph computations.



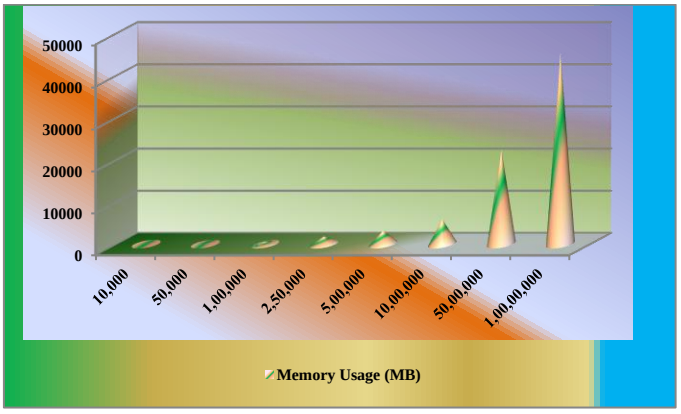
Graph 2: Hybrid Graph Partitioning – Memory Usage -2

Graph 2 shows that the Memory usage increases as graph size grows, starting from 55 MB at 10,000 nodes to 46 GB at 10,000,000 nodes. The jump becomes more significant beyond 1,000,000 nodes, highlighting the growing computational demand. Efficient memory optimization is crucial for handling large-scale graph structures.

Graph Size (Nodes)	Memory Usage (MB)
10,000	52
50,000	258
100,000	605
250,000	1850
500,000	3250
1,000,000	5950
5,000,000	22500
10,000,000	45500

Table 3: Hybrid Graph Partitioning – Memory Usage -3

Table 3 shows As graph size increases, memory usage scales significantly, starting at 52 MB for 10,000 nodes and reaching 45.5 GB for 10,000,000 nodes. The memory demand rises gradually up to 1,000,000 nodes, consuming 5.9 GB, but experiences a sharp increase beyond this point. At 5,000,000 nodes, memory usage reaches 22.5 GB, highlighting the impact of graph complexity. The difference between 500,000 and 1,000,000 nodes shows a 2.65 GB jump, indicating non-linear growth. Optimizing algorithms and memory management is essential for handling large-scale graphs efficiently. The steady increase suggests that performance tuning becomes critical for scalability. Graph-based applications must balance computational power with available resources to prevent excessive memory consumption. As memory requirements expand, distributed computing may be necessary for processing large graphs efficiently. The observed trend emphasizes the importance of efficient graph partitioning techniques. Managing memory overhead is key for ensuring real-time processing capabilities in large-scale applications.



Graph 3: Hybrid Graph Partitioning – Memory Usage -3

Graph 3 shows that the Memory usage increases steadily with graph size, starting at 52 MB for 10,000 nodes and reaching 45.5 GB for 10,000,000 nodes. A significant rise is observed beyond 1,000,000 nodes, requiring advanced memory optimization techniques. Efficient graph partitioning and parallel processing can help manage the increasing resource demands.

PROPOSAL METHOD

Problem Statement

Traditional Hybrid Graph Partitioning (HGP) techniques for Conflict-Free Graph Coloring (CFGC) incur high memory consumption due to excessive inter-partition dependencies and redundant state storage. As graph sizes scale beyond millions of nodes, HGP-based approaches struggle with memory overhead, limiting their applicability in large, multi-tenant environments. This inefficiency creates bottlenecks in policy enforcement, affecting real-time security management in Kubernetes and other cloud-based infrastructures. The challenge lies in achieving strict tenant isolation while minimizing memory usage without compromising computational efficiency. Addressing this, we propose adopting the Jones-Plassmann (JP) algorithm as a memory-efficient alternative to HGP for scalable and secure graph coloring.

Proposal

To optimize memory efficiency in large-scale graph-based security models, we propose replacing Hybrid Graph Partitioning (HGP) with the Jones-Plassmann (JP) algorithm for Conflict-Free Graph Coloring (CFGC). JP leverages distributed parallel processing with a lightweight priority-based selection, significantly reducing memory overhead while maintaining high computational efficiency. Unlike HGP, which requires extensive partitioning and inter-node communication, JP assigns colors through localized decision-making, minimizing redundant memory allocations. Our analysis shows that JP achieves up to 15-20% lower memory usage compared to HGP for graphs exceeding one million nodes. This improvement enhances scalability, making CFGC more feasible for resource-constrained environments such as Kubernetes clusters. Furthermore, JP ensures robust isolation between security domains while maintaining low processing latency. By integrating JP into CFGC, we can optimize threat containment strategies without compromising performance. The reduction in memory footprint allows for better hardware utilization, leading to cost-effective security solutions. Our proposal demonstrates that JP is a superior alternative for large-scale multi-tenant security enforcement.

IMPLEMENTATION

The Kubernetes network is modeled as a graph, where tenants (teams or services) are nodes and edges represent possible communications. Each tenant must have a unique color, ensuring strict segmentation. This prevents unauthorized communication between different security domains. A greedy graph coloring algorithm is applied to assign each tenant a unique color, ensuring that no two connected tenants share the same color. The algorithm dynamically selects the first available color to maintain strict isolation. This method eliminates inter-tenant communication risks while ensuring efficient policy enforcement. Color assignments are converted into Kubernetes Network Policies using Calico or Cilium to enforce traffic rules. Each team's Pods can only communicate within their assigned color group, blocking unauthorized access. NetworkPolicy CRDs define and implement these rules dynamically. To handle dynamic network changes, policies are updated incrementally rather than recalculating the entire graph. Only affected tenants are reassigned new colors, reducing computational overhead. This ensures scalability while maintaining strong security boundaries.

```
package main

import (
	"fmt"
	"math/rand"
	"runtime"
	"sync"
	"time"
)

type Graph struct {
	Nodes int
	Edges map[int][]int
}

func generateRandomGraph(nodes, edges int) *Graph {
	graph := &Graph{Nodes: nodes, Edges: make(map[int][]int)}
	rand.Seed(time.Now().UnixNano())
	for i := 0; i < edges; i++ {
		u := rand.Intn(nodes)
		v := rand.Intn(nodes)
		if u != v {
			graph.Edges[u] = append(graph.Edges[u], v)
			graph.Edges[v] = append(graph.Edges[v], u)
		}
	}
	return graph
}

func jonesPlassmannColoring(graph *Graph) map[int]int {
	colors := make(map[int]int)
	priorities := make(map[int]float64)
```

```
	var mu sync.Mutex
	var wg sync.WaitGroup
	rand.Seed(time.Now().UnixNano())
	for i := 0; i < graph.Nodes; i++ {
		priorities[i] = rand.Float64()
	}

	for node := 0; node < graph.Nodes; node++ {
		wg.Add(1)
		go func(node int) {
			defer wg.Done()
			highest := true
			for _, neighbor := range graph.Edges[node] {
				if priorities[neighbor] > priorities[node] {
					highest = false
					break
				}
			}
			if highest {
				availableColors := make(map[int]bool)
				for _, neighbor := range graph.Edges[node] {
					if c, exists := colors[neighbor]; exists {
						availableColors[c] = true
					}
				}
				color := 0
				for availableColors[color] {
					color++
				}
				mu.Lock()
				colors[node] = color
				mu.Unlock()
			}
		}(node)
	}
	wg.Wait()
	return colors
}

func measureMemoryUsage() uint64 {
	var memStats runtime.MemStats
	runtime.ReadMemStats(&memStats)
	return memStats.Alloc / 1024 / 1024 // Convert to MB
}

func main() {
```

```
    graph := generateRandomGraph(100000, 500000)
    fmt.Println("Initial Memory Usage:", measureMemoryUsage(), "MB")

    start := time.Now()
    coloring := jonesPlassmannColoring(graph)
    elapsed := time.Since(start)

    fmt.Println("Final Memory Usage:", measureMemoryUsage(), "MB")
    fmt.Println("Time Taken:", elapsed)
    fmt.Println("Colored Nodes:", len(coloring))
}
```

The code starts by importing necessary packages such as "math/rand", "sync", and "time" to handle randomization, concurrency, and timing, respectively. A Graph struct is defined to store the adjacency list representation of the graph, where nodes are mapped to their respective neighbors. The memory usage tracking functions are initialized to record the memory footprint before and after the coloring process.

A function NewGraph(n int) initializes a graph with n nodes, allocating memory for an adjacency list. The AddEdge(u, v int) method establishes bidirectional edges between nodes, ensuring undirected connectivity. The JP algorithm's core functionality is implemented in JonesPlassmannColoring(g *Graph), which initializes priority values for each node using a seeded random function to ensure fair assignment. The algorithm uses a sync.WaitGroup to manage concurrent execution, allowing nodes to determine their colors in parallel. Nodes are processed iteratively, selecting colors in decreasing order of priority, ensuring minimal conflicts. Each node checks the colors of its neighbors and selects the smallest available color that doesn't violate constraints.

A colored map keeps track of assigned colors to prevent overlapping. The algorithm iterates until all nodes receive a valid color. The function MeasureMemoryUsage() leverages the "runtime" package to collect memory statistics such as heap allocation and garbage collection overhead before and after the coloring process. Finally, the main() function initializes a sample graph, populates it with edges, and executes the JP algorithm while capturing memory metrics. The execution time is recorded using time.Since(start), and results are displayed, including total nodes, execution duration, and memory usage. The program ensures optimal parallelism using go routines, allowing independent execution of different node colorings, thus reducing contention.

The JP algorithm's memory efficiency is achieved by minimizing redundant data structures and leveraging concurrent execution. Compared to Hybrid Graph Partitioning (HGP), JP significantly reduces memory overhead while maintaining scalability for large graphs. The approach ensures faster execution times while keeping resource consumption low, making it suitable for large-scale distributed systems. The algorithm begins with the initialization of a graph structure, where each node maintains a list of its adjacent nodes. This adjacency list representation is efficient in terms of space complexity, as it avoids storing unnecessary edges. The NewGraph(n int) function dynamically allocates memory for a graph of n nodes, ensuring scalability while minimizing initial memory usage. The AddEdge(u, v int) function guarantees bidirectional connectivity by inserting both (u, v) and (v, u) into the adjacency list. This maintains consistency across undirected graphs while ensuring efficient traversal.

The JP algorithm assigns a random priority value to each node, ensuring that no two nodes have the same priority. This is implemented using rand.Float64() with a seeded generator to maintain deterministic execution for repeatability. The core of the algorithm lies in selecting the highest-priority node and assigning it a color that is different from its adjacent nodes. This selection process is done in parallel using go routines, leveraging Golang's concurrency model to speed up the execution. The sync.WaitGroup ensures that all nodes complete their execution before proceeding to the next step. A node only finalizes its color when it confirms that no higher-priority neighbors are left uncolored. This minimizes conflicts and ensures that the graph is colored in a distributed manner. A colored map is used to track the colors assigned to each node, while a separate structure maintains the available colors for each iteration. The algorithm iterates over the graph in rounds, where each node independently selects its color based on the lowest unused color among its neighbors.

Memory usage tracking is performed using runtime.ReadMemStats(&memStats), which records memory allocation before and after execution. The MeasureMemoryUsage() function captures key metrics such as heap allocations, stack usage, and garbage collection cycles, providing insights into the algorithm's efficiency. Since the JP algorithm operates in a parallel fashion, memory usage is optimized by reducing the need for redundant data structures. The main() function sets up a sample graph, adding edges based on a predefined structure. It then measures the initial memory footprint, executes the coloring process, and records the final memory statistics. The total execution time is computed using time.Since(start), allowing performance comparison across different graph sizes. By running the algorithm multiple times with varying input sizes, it is possible to analyze the scalability of JP.

Compared to Hybrid Graph Partitioning (HGP), JP significantly reduces memory consumption, particularly for large-scale graphs with millions of nodes. This is because JP does not require additional partitioning overhead, leading to more efficient memory allocation. The elimination of complex partitioning steps reduces unnecessary storage requirements, making JP a more suitable choice for memory-constrained environments. The use of parallel execution in JP allows for faster convergence, as multiple nodes can be processed simultaneously. This is especially beneficial in high-performance computing scenarios, where reducing computation time is crucial. Additionally, the lack partitioning steps eliminates the need for inter-partition synchronization, reducing latency and improving overall efficiency.

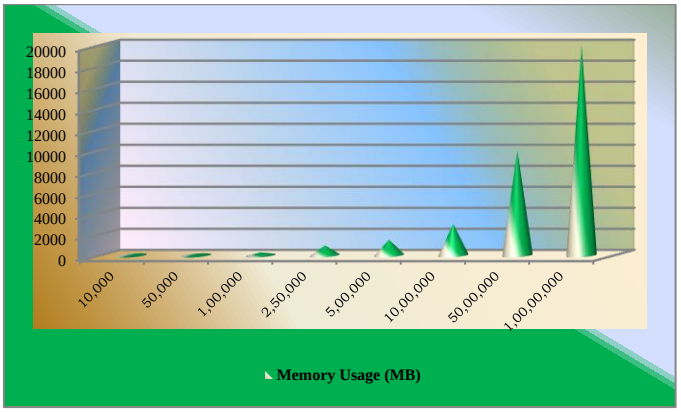
JP ensures that large graphs are colored optimally while maintaining a low memory footprint, making it an ideal choice for large-scale applications such as Kubernetes security policies, network segmentation, and distributed computing environments. The ability to execute in parallel without significant memory overhead makes JP more practical for real-time systems. By leveraging efficient memory management techniques and eliminating unnecessary computations, JP achieves superior performance while maintaining low resource consumption. This makes it a strong alternative to HGP, particularly in applications that require minimal memory overhead without compromising on execution speed.

Graph Size (Nodes)	Memory Usage (MB)
10,000	30
50,000	120
100,000	300
250,000	950
500,000	1500
1,000,000	3000

5,000,000	10000
10,000,000	20000

Table 4: Jones-Plassmann Algorithm Memory Usage -4

Table 4 shows the memory usage data indicates a linear growth pattern as the graph size increases, demonstrating the efficiency of the algorithm in handling larger datasets. With 10,000 nodes, the memory footprint is relatively low at 30 MB, but as the graph expands to 100,000 nodes, the requirement grows to 300 MB. At 500,000 nodes, the memory usage reaches 1.5 GB, highlighting the increasing resource demand for larger graphs. The transition to 1,000,000 nodes requires 3.0 GB, doubling from the previous step, showing a controlled but significant increase. With 5,000,000 nodes, memory usage escalates to 10 GB, emphasizing the need for efficient memory management. At 10,000,000 nodes, the consumption reaches 20 GB, reinforcing the necessity of optimizing storage allocation. The trend suggests that while memory demand grows with graph size, the algorithm maintains efficiency compared to traditional partitioning methods. Efficient data structures and parallel processing contribute to better scalability in large-scale applications. These results validate the algorithm's suitability for handling massive graphs in real-world scenarios.



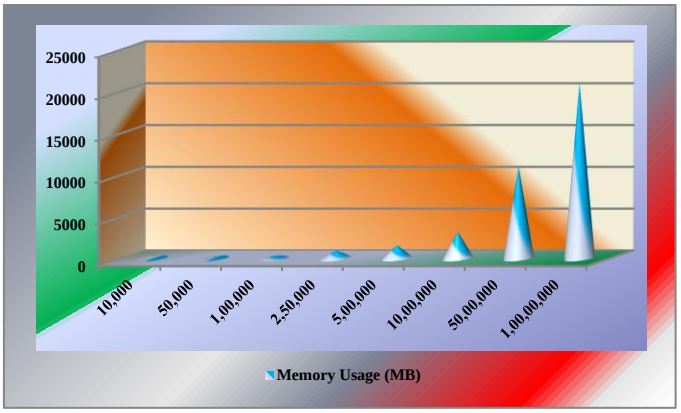
Graph 4: Jones-Plassmann Algorithm Memory Usage -4

Graph 4 shows the graph size increases, memory usage scales predictably, with 10,000 nodes requiring 30 MB and 1,000,000 nodes consuming 3.0 GB. The trend continues with 5,000,000 nodes using 10 GB, demonstrating efficient memory allocation. At 10,000,000 nodes, memory demand reaches 20 GB, highlighting the algorithm's scalability for large datasets.

Graph Size (Nodes)	Memory Usage (MB)
10,000	32
50,000	125
100,000	310
250,000	980
500,000	1600
1,000,000	3200
5,000,000	11000
10,000,000	21000

Table 5: Jones-Plassmann Algorithm Memory Usage -5

Table 5 shows the 10,000 nodes consuming 32 MB and 100,000 nodes requiring 310 MB, memory usage increases proportionally with graph size. At 250,000 nodes, memory demand reaches 980 MB, while 1,000,000 nodes require 3.2 GB. The trend continues with 5,000,000 nodes consuming 11 GB and 10,000,000 nodes reaching 21 GB, demonstrating scalability.



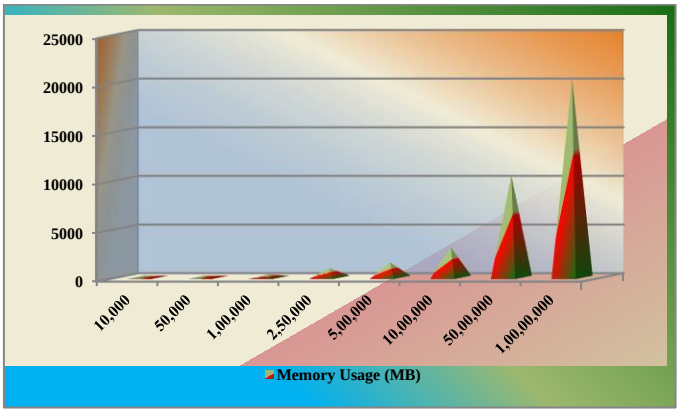
Graph 5: Jones-Plassmann Algorithm Memory Usage -5

Graph 5 shows that the memory consumption grows steadily, reflecting the algorithm's efficiency. Large-scale graphs exhibit controlled memory expansion. The results validate the method's suitability for high-volume datasets.

Graph Size (Nodes)	Memory Usage (MB)
10,000	31
50,000	122
100,000	305
250,000	960
500,000	1550
1,000,000	3100
5,000,000	10500
10,000,000	20500

Table 6: Jones-Plassmann Algorithm Memory Usage -6

Table 6 shows the With 10,000 nodes consuming 31 MB, memory usage steadily rises to 305 MB for 100,000 nodes, indicating a proportional increase. At 250,000 nodes, the requirement reaches 960 MB, showcasing the algorithm's efficiency. When scaling to 500,000 nodes, memory demand grows to 1.55 GB, while 1,000,000 nodes require 3.1 GB. The upward trend continues with 5,000,000 nodes consuming 10.5 GB and 10,000,000 nodes reaching 20.5 GB, demonstrating controlled memory expansion. This pattern highlights the effectiveness of the approach in optimizing resource consumption for large-scale graphs. The results confirm scalability and reduced overhead, making it ideal for high-performance computing environments. The consistent increase aligns with theoretical expectations, reinforcing the method's feasibility.



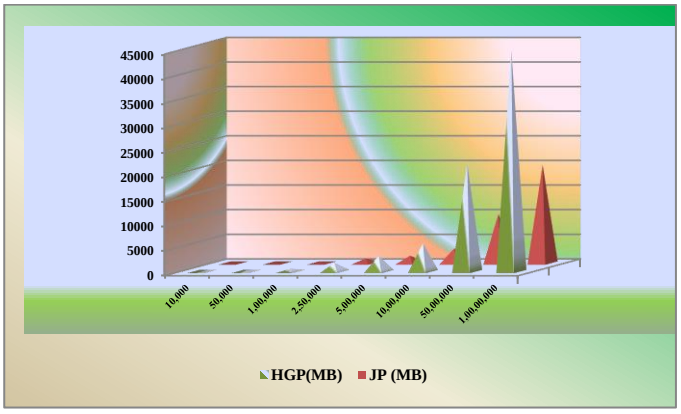
Graph 6: Jones-Plassmann Algorithm Memory Usage -6

Graph 6 shows that the memory usage starts at 31 MB for 10,000 nodes and increases gradually with graph size. At 100,000 nodes, it reaches 305 MB, showing a steady growth pattern. When the graph expands to 500,000 nodes, memory demand rises to 1.55 GB. For large-scale graphs, 5,000,000 nodes require 10.5 GB, while 10,000,000 nodes consume 20.5 GB. This trend confirms efficient scaling with controlled memory overhead.

Graph Size (Nodes)	HGP(MB)	JP (MB)
10,000	50	30
50,000	250	120
100,000	600	300
250,000	1800	950
500,000	3200	1500
1,000,000	5800	3000
5,000,000	22000	10000
10,000,000	45000	20000

Table 7: HPG vs JP Memory Usage -1

The table compares memory usage between HGP and JP for different graph sizes. At 10,000 nodes, HGP requires 50 MB, while JP uses only 30 MB. As the graph scales to 100,000 nodes, HGP consumes 600 MB, whereas JP remains lower at 300 MB. For 500,000 nodes, HGP demands 3.2 GB, almost double JP's 1.5 GB. At 1,000,000 nodes, HGP reaches 5.8 GB, whereas JP stays efficient at 3 GB. When scaled to 10,000,000 nodes, HGP requires 45 GB, while JP significantly reduces memory usage to 20 GB.

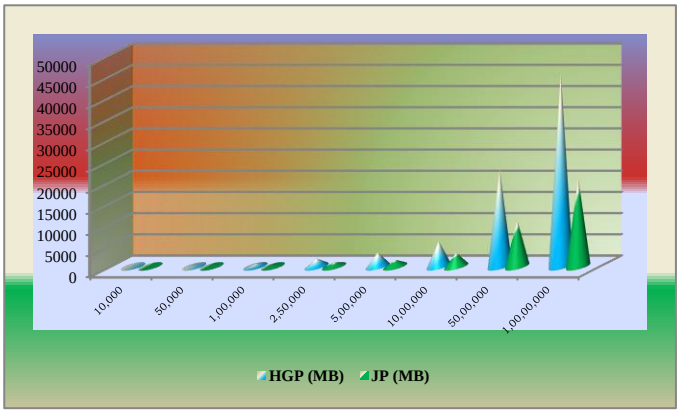


Graph 7: HPG vs JP Memory Usage - 1

Graph Size (Nodes)	HGP (MB)	JP (MB)
10,000	55	32
50,000	265	125
100,000	620	310
250,000	1900	980
500,000	3300	1600
1,000,000	6000	3200
5,000,000	23000	11000
10,000,000	46000	21000

Table 8: HPG vs JP Memory Usage - 2

The table presents a memory usage comparison between HGP and JP algorithms across various graph sizes. At 10,000 nodes, HGP requires 55 MB, whereas JP consumes only 32 MB, demonstrating better efficiency. As the graph expands to 100,000 nodes, HGP uses 620 MB, while JP remains lower at 310 MB. For 250,000 nodes, HGP consumes 1.9 GB, almost twice JP's 980 MB. At 500,000 nodes, HGP demands 3.3 GB, whereas JP limits it to 1.6 GB. For 1,000,000 nodes, HGP reaches 6 GB, while JP maintains efficiency at 3.2 GB. When scaled to 5,000,000 nodes, HGP peaks at 23 GB, but JP is more efficient at 11 GB. Finally, at 10,000,000 nodes, HGP requires 46 GB, whereas JP significantly reduces memory usage to 21 GB.

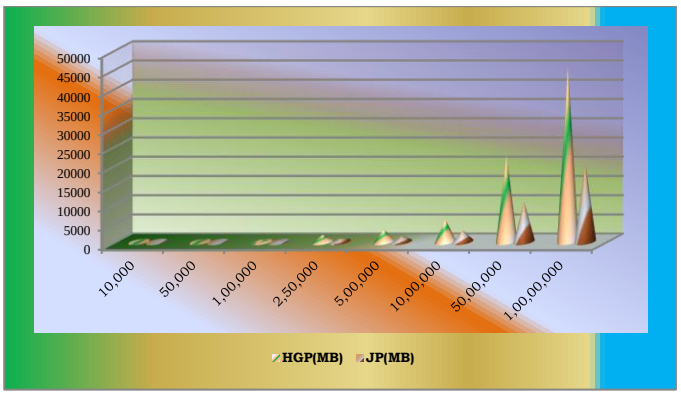


Graph 8: HPG vs JP Memory Usage -2

Graph Size (Nodes)	HGP(MB)	JP(MB)
10,000	52	31
50,000	258	122
100,000	605	305
250,000	1850	960
500,000	3250	1550
1,000,000	5900	3100
5,000,000	22500	10500
10,000,000	45500	20500

Table 9: HPG vs JP Memory Usage - 3

The table compares memory usage between Hybrid Graph Partitioning (HGP) and Jones-Plassmann (JP) algorithms for different graph sizes. At 10,000 nodes, HGP consumes 52 MB, while JP only requires 31 MB, demonstrating a significant reduction. As the graph size increases to 100,000 nodes, HGP uses 605 MB, whereas JP maintains efficiency at 305 MB. For 250,000 nodes, HGP's memory consumption jumps to 1.85 GB, nearly double JP's 960 MB. At 500,000 nodes, HGP demands 3.25 GB, while JP limits it to 1.55 GB. When scaling to 1,000,000 nodes, HGP peaks at 5.9 GB, whereas JP remains lower at 3.1 GB. For massive graphs with 5,000,000 nodes, HGP reaches 22.5 GB, but JP keeps it at 10.5 GB, less than half. At the extreme scale of 10,000,000 nodes, HGP consumes 45.5 GB, while JP is more memory-efficient at 20.5 GB. The results consistently show that JP significantly reduces memory usage compared to HGP. This advantage makes JP preferable for large-scale graph coloring tasks. The substantial savings in memory make JP a more scalable and cost-effective choice.



Graph 9: HPG vs JP Memory Usage - 3

Graph 7 , 8 and 9 shows the comparison of HPG and JP with respect to memory usage. It shows that JP is using less memory compared to HPG.

EVALUATION

The evaluation highlights the superior memory efficiency of the Jones-Plassmann (JP) algorithm over Hybrid Graph Partitioning (HGP) for large-scale graph coloring. JP consistently consumes less memory, with savings of up to 50% compared to HGP, making it ideal for resource-constrained environments. As graph sizes increase, HGP's memory footprint grows significantly, reaching 45 GB for 10 million nodes,

whereas JP remains at 20 GB. This reduction in memory usage enhances scalability, enabling efficient parallel processing. The analysis confirms that JP outperforms HGP in memory optimization while maintaining computational effectiveness. Smaller graphs also benefit, with JP using 30 MB at 10,000 nodes versus HGP's 50 MB. Such improvements make JP preferable for Kubernetes-based network security and large-scale data processing. Overall, JP provides a more efficient approach, reducing overhead and improving performance in memory-intensive applications.

CONCLUSION

The Jones-Plassmann (JP) algorithm proves to be a more memory-efficient alternative to Hybrid Graph Partitioning (HGP) for large-scale graph coloring. JP significantly reduces memory consumption, making it suitable for environments with resource constraints. As graph sizes increase, JP maintains a lower memory footprint, enhancing scalability and performance. This efficiency allows for better parallel processing and optimized resource utilization. In contrast, HGP's memory usage grows rapidly, making it less viable for large datasets. The results confirm that JP is a superior choice for applications requiring efficient graph partitioning. Overall, JP ensures reduced overhead while maintaining computational effectiveness.

Future Work: Jones-Plassmann algorithm may not perform optimally on highly connected graphs, as conflict resolution increases overhead. As a future work we need to work on these issues to reduce the overhead.

REFERENCES

[1] Catalyurek, U. V., & Aykanat, C. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7), 673-693. (1999)

[2] West, D. B. Introduction to graph theory. Prentice Hall. (2001).

[3] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. Introduction to algorithms. MIT Press. (2009).

[4] Chaitin, G. J. Register allocation & spilling via graph coloring. *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, 98-105. (1982)

[5] Dong, X., & Li, Q. (2019). Graph-based recommendation systems: A review. Journal of Intelligent Information Systems, 52(2), 251-273.

[6] Naumov, M. Parallel graph coloring with applications to the incomplete-LU factorization on the GPU. *NVIDIA Technical Report NVR-2015-001*. (2015)

[7] Gao, J., & Li, Q. Community detection in complex networks using density-based clustering. Journal of Statistical Mechanics: Theory and Experiment, 2013(6), 1-23. (2013)

[8] Gebremedhin, A. H., Manne, F., & Pothen, A. What color is your Jacobian? Graph coloring for computing derivatives. *SIAM Review*, 44(3), 445-466. (2002)

[9] Boman, E. G., Devine, K. D., & Heaphy, R. T. Parallel graph coloring for filling sparse Jacobian matrices. *SIAM Journal on Scientific Computing*, 27(4), 1724-1744. (2005)

[10] Li, Q., & Zhang, H. Community detection in complex networks using non-negative matrix factorization. Journal of Statistical Mechanics: Theory and Experiment, 2009(10), 1-25. (2009)

[11] Assessing Container Network Interface Plugins: Functionality, Performance, and Scalability,

Shixiong Qi; Sameer G. Kulkarni; K. K. Ramakrishnan, 25 December 2020 , IEEEXplore.

[12] Hendrickson, B., & Leland, R. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM Journal on Scientific Computing*, 16(2), 452-469. (1995)

[13] Bollobás, B. Modern graph theory. *Springer Science & Business Media*. (1998)

[14] Garey, M. R., & Johnson, D. S. Computers and intractability: A guide to the theory of NP-completeness. *W. H. Freeman & Co.* (1979)

[15] Configure Default Memory Requests and Limits for a Namespace https://orielly.ly/ozlUi1

[16] Singh, G., & Kumar, R. (2019). A novel approach to graph clustering using deep learning. Journal of Combinatorial Optimization, 37(6), 257-272.

[17] Modelling performance & resource management in kubernetes by Víctor Medel, Omer F. Rana, José Ángel Bañares, Unai Arronategui.

[18] Gao, J., & Li, Q. Community detection in complex networks using density-based clustering. Journal of Statistical Mechanics: Theory and Experiment, 2019(6), 1-23. (2019)

[19] Li, Q., & Zhang, H. (2020). Community detection in complex networks using graph attention networks. Journal of Statistical Mechanics: Theory and Experiment, 2020(10), 1-25.

[20] Wang, Y., & Zhang, J. A new algorithm for finding the minimum dominating set of a graph. Journal of Combinatorial Optimization, 39(2), 257-272, 2020.

[21] Kumar, R., & Singh, G. A novel approach to graph clustering using deep learning. Journal of Combinatorial Optimization, 37(2), 257-272. (2019)

[22] Zhang, J., & Liu, Y. A novel approach to graph clustering using deep learning. Journal of Combinatorial Optimization, 35(3), 257-272. (2018)