Bangladesh University of Engineering and Technology (BUET), Dhaka, Bangladesh

Department of Computer Science and Engineering

GRAPH PARTITIONING

Course: CSE - 462 (Algorithm Engineering Sessional)

1905064 - Sayem Shahad
1905065 - Arnab Bhattacharjee
1905079 - Salman Sayeed
1905089 - Majisha Jahan Disha
1905109 - Arko Sikder

December 27, 2024

Contents

1 Introduction

Graph partitioning is a fundamental concept in graph theory and computer science. It involves dividing the set of vertices V of a graph $G = (V, E)$ into disjoint subsets such that specific criteria are met. Depending on the application, graph partitioning can focus on unbalanced or balanced partitions, and it plays a vital role in areas like parallel computing, data clustering, and VLSI design.

1.1 Basic Graph Partitioning

The essence of graph partitioning lies in dividing the vertex set V into subsets such that the number of edges (known as the *cut size*) between the subsets is minimized. The partitions can be:

- **Unbalanced Partition:** No constraints on the sizes of the subsets.
- **Balanced Partition:** Subsets must have nearly equal sizes, ensuring computational or workload balance.

The visualization in Figure 1 illustrates the difference between an unbalanced and a balanced partition.

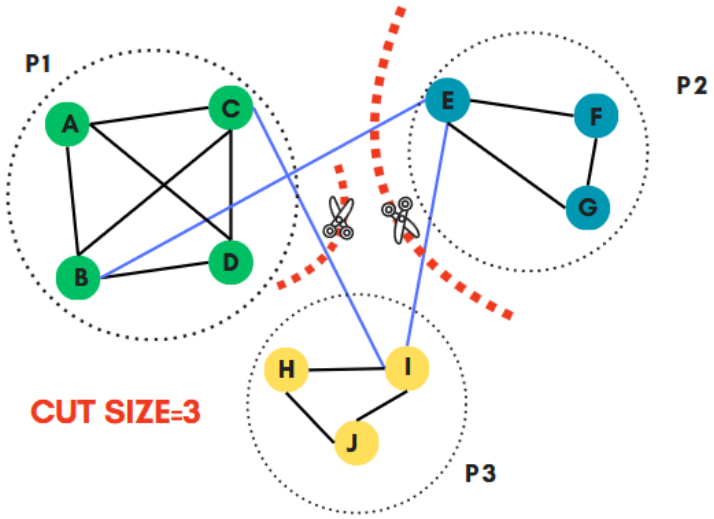

Figure 1: Unbalanced vs. Balanced Graph Partitioning

1.2 The Graph Partitioning Problem (GPP)

The **Graph Partitioning Problem (GPP)** extends basic graph partitioning by introducing additional constraints and objectives. It is formally defined as follows:

- **Input:** An undirected graph $G = (V, E)$ and an integer k, the number of desired partitions.
- **Output:** A partition of V into k subsets $V_1, V_2, \ldots, V_k$ such that:
  - The subsets are disjoint: $V_i \cap V_j = \emptyset$ for $i \neq j$.
  - All vertices are assigned: $\bigcup_{i=1}^{k} V_i = V$.
  - The sizes of the subsets satisfy the balance constraint: $|V_i| \leq 1 + \lceil |V|/k \rceil$ for all i.
  - The number of edges crossing between partitions (*cut size*) is minimized: $cut(V_1, \ldots, V_k) = \sum_{(u,v)\in E} \mathbb{I}[u \in V_i, v \in V_j, i \neq j]$.

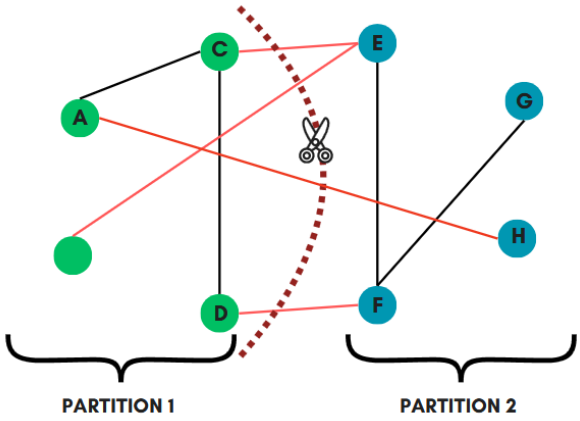

Figure 2: An example of the Graph Partitioning Problem for $k = 2$

The GPP is a well-known **NP-hard** optimization problem for $k \geq 2$, whether the graph is weighted or unweighted. Its complexity arises from the exponential growth of possible partitions as the graph size increases.

The key challenges in solving the GPP include:

- Balancing partition sizes while minimizing cut size.
- Ensuring scalability for large graphs.
- Designing efficient algorithms for practical use cases.

2 Polynomial Time Reduction

The Graph Partitioning Problem (GPP) is an optimization problem where the objective is to partition the vertex set V of a graph $G = (V, E)$ into disjoint subsets such that the cut size is minimized, subject to balance constraints. To prove that GPP is NP-hard, we need to establish the NP-completeness of its decision version.

2.1 Decision Version of GPP

The decision version of the Graph Partitioning Problem is defined as follows:

- **Input:** An undirected graph $G = (V, E)$, an integer k, and the desired number of partitions p.
- **Question:** Can the vertex set V be partitioned into p subsets $V_1, V_2, \ldots, V_p$ such that:
  1. $\bigcup_{i=1}^{p} V_i = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$.
  2. Each subset satisfies the balance constraint: $|V_i| \leq 1 + \lceil |V|/p \rceil$.
  3. The number of edges crossing between the subsets is at most k.

This version is classified as NP-complete because:

1. It belongs to NP: A ”yes” answer can be verified in polynomial time by checking the partitioning and counting the cut size.
2. It can be reduced from a known NP-complete problem, as shown below.

2.2 Reduction from Simple Max Cut Problem

To prove NP-completeness, we reduce the **Simple Max Cut Problem** (a known NP-complete problem) to the decision version of GPP.

2.2.1 Simple Max Cut Problem Definition

The Simple Max Cut Problem is defined as follows:

- **Input:** A graph $G = (V, E)$ and an integer k.
- **Question:** Is there a partition of V into two subsets V_1 and V_2 such that the number of edges crossing between the subsets (*cut size*) is at least k?

2.2.2 Reduction Process

Given an instance of the Simple Max Cut Problem, $G = (V, E)$ and k:

1. Create a new graph $G^* = (V^*, E^*)$ by:
   - Adding n new vertices $u_1, u_2, \ldots, u_n$ to V, resulting in $V^* = V \cup \{u_1, u_2, \ldots, u_n\}$.
   - Defining E^* as the complement of E, such that $E^* = \{(v, v') \mid v, v' \in V^*, (v, v') \notin E\}$.
2. Set the target cut size $k^* = n^2 - k$, where $n = |V|$.
3. Formulate the decision problem for G^*: Can V^* be partitioned into two balanced subsets W_1 and W_2 such that the cut size is $\leq k^*$?

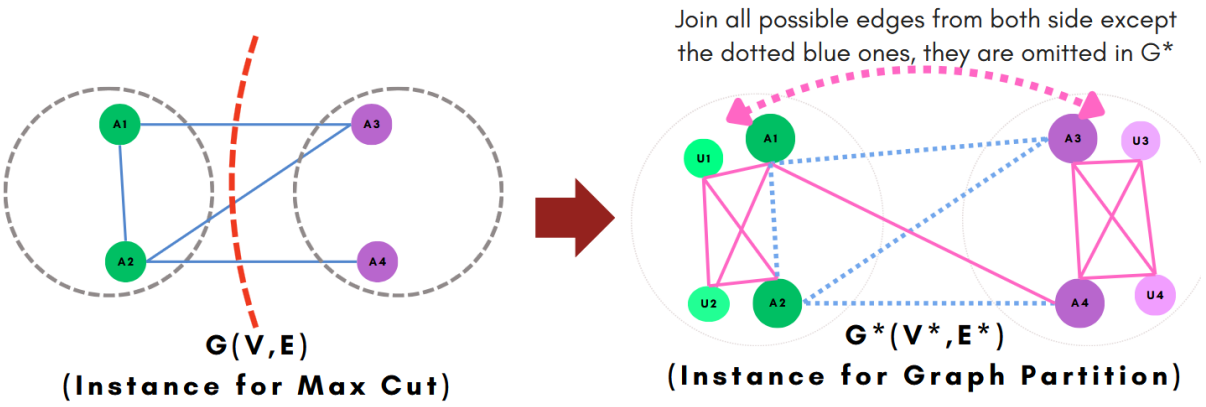

Figure 3: Reduction of a Simple Max Cut instance to a Graph Partitioning instance.

2.2.3 Proof of Correctness

We now prove the correctness of this reduction:

Simple Max Cut $\rightarrow$ GPP Decision Version: If the Simple Max Cut Problem (G, k) has a "yes" solution, then the GPP instance (G^*, k^*) also has a "yes" solution:

Edges in E^* between partitions W_1 and $W_2 = n^2 -$ edges in E between V_1 and $V_2 \leq n^2 - k = k^*$.

Thus, a valid Max Cut in G corresponds to a valid partition in G^*, satisfying the cut size constraint.

GPP Decision Version → Simple Max Cut: If the GPP instance (G^*, k^*) has a "yes" solution, then the Simple Max Cut instance (G, k) also has a "yes" solution:

$$\text{Edges in } E \text{ between partitions } V_1 \text{ and } V_2 = n^2 - \text{edges in } E^* \text{ between } W_1 \text{ and } W_2 \geq n^2 - k^* = k.$$

Thus, a valid partition in G^* implies a valid Max Cut in G.

2.3 Conclusion

This reduction demonstrates that solving the decision version of GPP is equivalent to solving the Simple Max Cut Problem. Since Simple Max Cut is NP-complete, the decision version of GPP is also NP-complete, and the optimization version of GPP is therefore NP-hard.

3 Variants

The Graph Partitioning Problem (GPP) encompasses multiple variants, each addressing specific constraints and optimization goals. These variants are tailored to meet the demands of diverse applications, ranging from load balancing to network optimization.

3.1 Unbalanced Partitioning (Unconstrained)

In unbalanced partitioning, there are no restrictions on the sizes of the subsets. The sole objective is to minimize the number of edges (cut size) between partitions. This variant is suitable for scenarios where the sizes of the partitions are not a critical factor.

3.2 Balanced Partitioning (Constrained)

Balanced partitioning imposes size constraints on the subsets. Each partition V_i must satisfy the balance condition $|V_i| \leq 1 + \lceil |V|/k \rceil$. The goal is to minimize the cut size while ensuring that the subsets are nearly equal in size. Balanced partitioning is widely used in parallel computing and task allocation problems, where workload distribution plays a crucial role.

3.3 r-Bound Partitioning (Constrained)

The r-Bound Partitioning variant introduces additional restrictions on the partition sizes or other specific properties. For instance, partitions might need to satisfy particular bounds on the number of vertices or edges. This variant is often applied in hierarchical graph structures or problems requiring strict adherence to size limits.

3.4 Multi-Level Partitioning

Multi-level partitioning involves a three-step process:

1. **Coarsening:** The graph is reduced by merging nodes and edges, creating a simpler representation.
2. **Partitioning:** The reduced graph is partitioned using standard algorithms.
3. **Uncoarsening:** The solution is refined by projecting the partitioning back onto the original graph and improving it iteratively.

This approach is highly efficient for handling large graphs and produces high-quality partitions.

3.5 Weighted Graph Partitioning

In weighted graph partitioning, weights are assigned to vertices or edges to represent additional information such as computational load or communication costs. The objective is to minimize the weighted cut size while maintaining balance constraints. This variant is critical in resource allocation and traffic management problems.

Our chosen variant

In the subsequent sections, we will delve deeper into the Balanced Partitioning variant. The following discussions will include common approaches for solving it, as it forms the foundation for many practical applications of the Graph Partitioning Problem.

4 Heuristic Algorithm

4.1 Kernighan-Lin algorithm

Initially, let us suppose that we are looking for a perfectly balanced bisection (i.e., of unit partitioning balance) and that all the vertices have the same weight. We will see in the following sections the way to adapt the algorithm to parts of different sizes.

To generate the two subsets of the bisection to exchange, the authors have introduced two notions: the **external cost** and the **internal cost**. The internal cost $I(v_i)$ of a vertex v_i of the part V_i of the bisection is defined as the sum of the weights of the edges between v_i and the vertices that do belong to V_i:

$$I(v_i) = \sum_{v'_i \in V_i} w(v_i, v'_i) \tag{1}$$

The **external cost** $E(v_i)$ of a vertex v_i of V_i is defined as the sum of the weights of the edges between v_i and the vertices that do not belong to V_i:

$$E(v_i) = \sum_{v'_i \in V \setminus V_i} w(v_i, v'_i) \tag{2}$$

Let $D(v_i)$ be the difference between the external cost and the internal cost of the vertex v_i:

$$D(v_i) = E(v_i) - I(v_i) \tag{3}$$

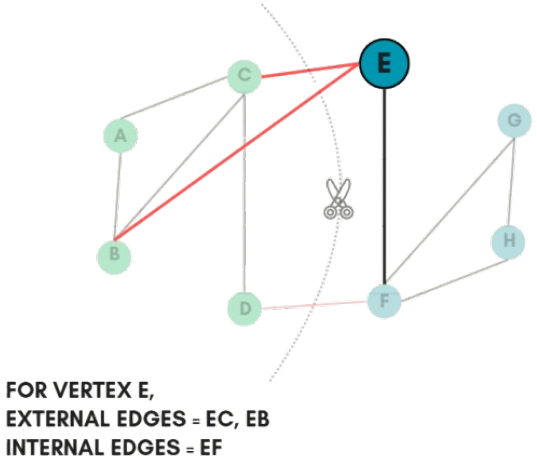

Figure 4: the external and internal edges of a vertex for cut calculation.

Consider two vertices, each belonging to a different part of the bisection $P = (V_1, V_2)$, $v_1 \in V_1$ and $v_2 \in V_2$. The **gain**, denoted by g, resulting from the exchange of v_1 and v_2 is:

$$g(v_1, v_2) = D(v_1) + D(v_2) - 2w(v_1, v_2) \tag{4}$$

For each vertex $v_i \in V_i$, the algorithm calculates the difference between the external cost and the internal cost of v_i, that is $D(v_i)$. The values $D(v_i)$ are distributed into two sets D_1 and D_2, according to the part which v_i belongs to. Among all pairs, choose the one with the maximum gain. and then lock.

Algorithm 1 The Kernighan-Lin algorithm

procedure KL(graph $G = (V, E)$, bisection $P_2 = (V_1, V_2)$ of G)
    $cut \leftarrow cut(P_2)$
    repeat ▷ "pass" is also called *phase 1 optimization*
        Compute D_1 and D_2
        for $q = 1$ to $|V|/2$ **do**
            Choose $v_1^q \in V_1^q$ and $v_2^q \in V_2^q$ that maximize the gain g
            $E_1(q) \leftarrow V_1^q$, $E_2(q) \leftarrow V_2^q$, $G(q) \leftarrow g$
            Lock v_1^q and v_2^q
            Update D_1 and D_2
        Choose q that maximizes G
        if $G(q) > 0$ **then**
            $cut \leftarrow cut + G(q)$
            Exchange $E_1(q)$ and $E_2(q)$ in P_2
    until $G(q) \leq 0$
    return (P_2, cut)

4.2 Fiduccia-Mattheyses implementation

Algorithm 2 Fiduccia-Mattheyses algorithm adapted to graph partitioning

procedure FM(graph $G = (V, E)$, bisection P_2, max balance $maxbal$ of P_2)
    $cut \leftarrow cut(P_2)$
    repeat ▷ "pass" is also called *phase 1 optimization*
        Create the two gains tables
        for $i = 1$ to $|V|$ **do**
            Select an unlocked vertex $v \in V$ such that its gain is maximal and after transfer the balance $maxbal$ is observed
            for all unlocked vertex v' adjacent to v **do**
                Update the gains table for v'
            Lock v
        Select i that maximizes the gain g
        if $g(i) > 0$ **then**
            Alter the bisection P_2 according to i
            $cut \leftarrow cut + g(i)$
    until $g(i) \leq 0$
    return (P_2, cut)

FM algorithm has two efficiency update.

4.2.1 Efficient gain update with gain table

The data structure that can maintain the gain of the vertices updated is shown in Figure 5. This data structure consists of maintaining, for each part of the bisection, a sorted table of the gains of the vertices.A doubly-linked list is associated with each cell of the gains table. This doubly-linked list contains the vertices whose gain corresponds to the rank of the list in this gains table. Finally, each cell of the vertices table points at its vertex, and belongs to one of the doubly-linked lists. Thus, accessing a vertex runs in constant time, and its transfer from one gain to another during the gains update also runs in constant time.

- Gain will be updated for the node that is moved and its neighbours.
- Neighbours will be shifted up and down. The neighbors from the new partition will be shifted down by 2 cells in the gain table (their gains will be reduced by 2). The neighbors from the old partition will be shifted up by 2 cells in the gain table (their gains will be updated by 2)
- Gain for the shifted node should be recalculated.

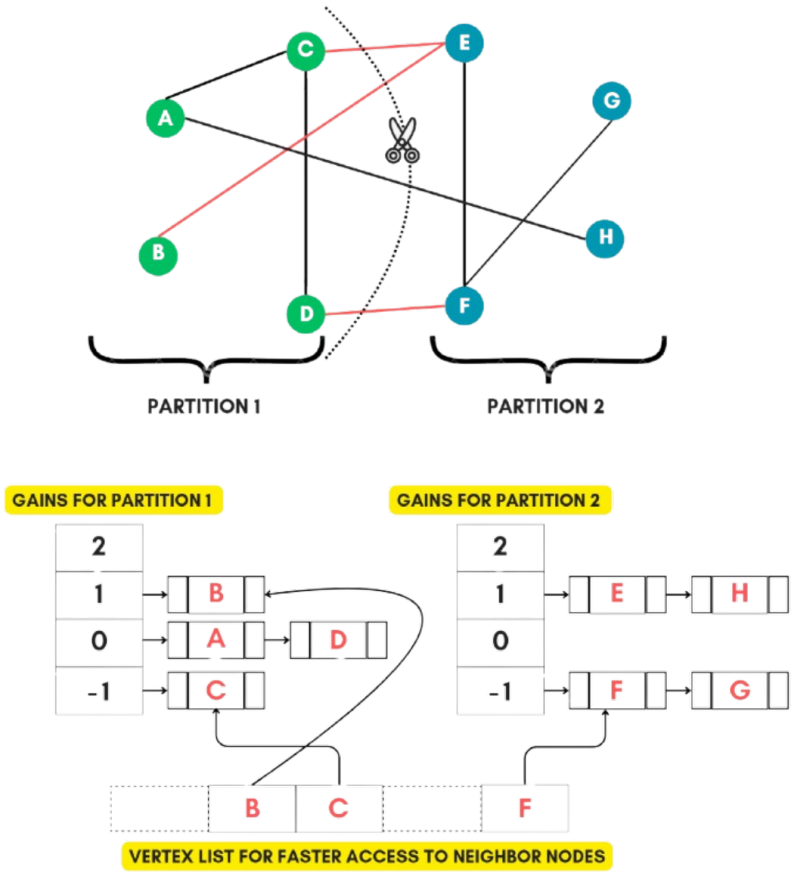

Figure 5: gain tables for FM algorithm

4.3 Global Kernighan-Lin Refinement

The refinement algorithm Global Kernighan-Lin Refinement (GKLR) uses the same data structure as the one from the Fiduccia-Mattheyses implementation, which can maintain the vertices gain updated, the only difference being that only one gains table is used. As all the Kernighan-Lin algorithms, this algorithm is iterative and each iteration is called a “pass”. Each pass consists of moving a set of vertices to other parts in order to reduce the cut of the partition.

Algorithm 3 The Global Kernighan-Lin Refinement algorithm

procedure GKLR(graph $G = (V, E)$, partition P_k, max balance $maxbal$)
  repeat ▷ "pass"
    Create the gains table and the set of unlocked vertices V_v
    while $V_v \neq \emptyset$ **or** dep gains have not been negative **do**
      Select the vertex v of greatest gain. Let $V_j \in P_k$ such that $v \in V_j$
      for all $V_i \in P_k - \{V_j\}$ such that $V_i \cup \{v\}$ maintains $maxbal$ **do**
        Compute the gain of adding v to V_i
      Select the part $V_i \in P_k$ of maximal gain g for the vertex v
      Record the transfer of v to V_i and the corresponding gain g
      for all unlocked vertex v' adjacent to v **do**
        Update the gains table for v'
      Lock v
    Select the set of vertices to transfer that maximizes the gain g
    if $g > 0$ **then**
      Alter P_k by adding the set of vertices to transfer
  until $g \leq 0$ **or** a predefined number of passes is reached
  return P_k

Once this iteration is complete, the set of vertices, if any, that has allowed to reduce most of the cut, is moved to the corresponding parts. The move of these vertices ends the pass. The iteration on the passes ends when a predefined number of passes have been executed or when a pass has not reduced the cut of the partition. One of the main drawbacks of this refinement algorithm is that it is difficult to correctly refine partitions that do not already maintain the partitioning balance. This algorithm is often used in multilevel methods. However, the uncoarsening and refinement phase of the multilevel methods does not always produce a partition maintaining the partitioning balance.

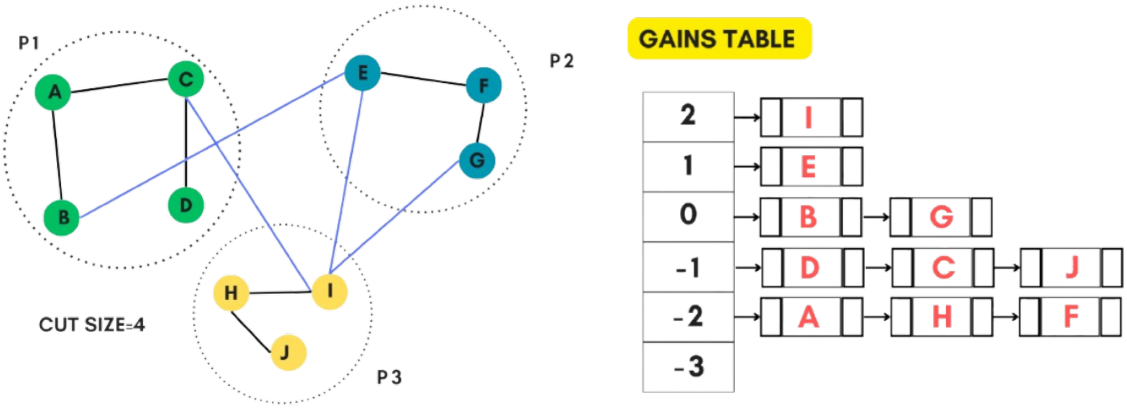

Figure 6: Just maintain one gain table for all partitions. Choose I, the node with the maximum gain from the gain table

In figures 7, 8, and 9 the updates in neighbor nodes are shown.

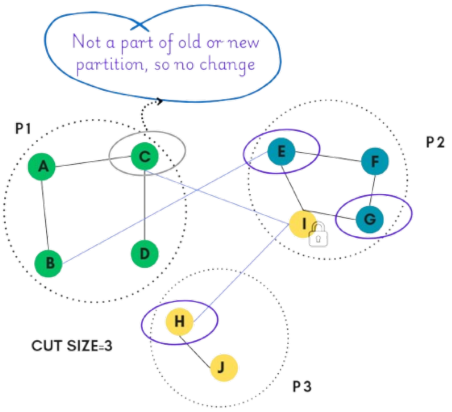

Figure 7: For some neighbors, gain might not be updated

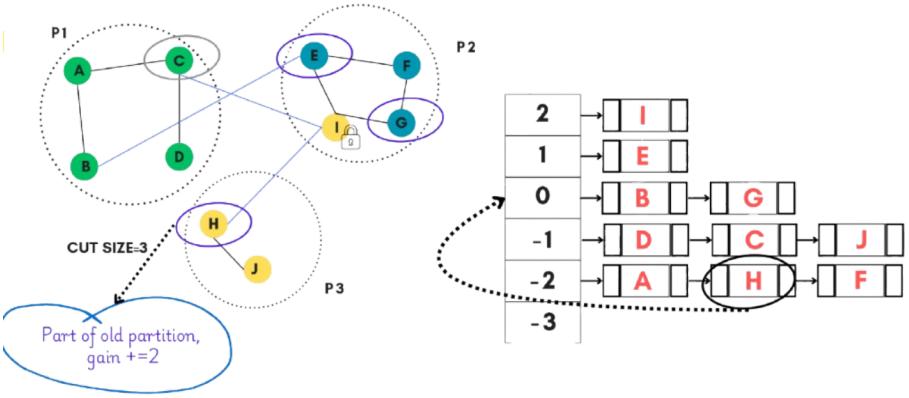

Figure 8: The neighbors from the old partition will be shifted up by 2 cells in the gain table (their gains will be updated by 2)

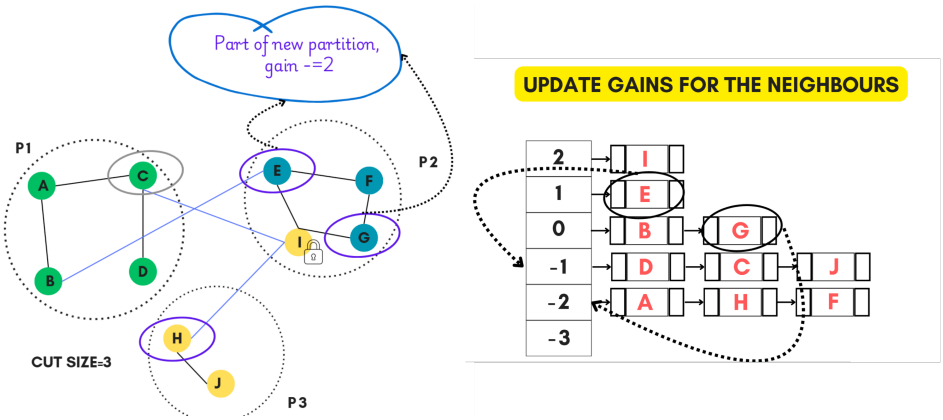

Figure 9: The neighbors from the new partition will be shifted down by 2 cells in the gain table (their gains will be reduced by 2)

some more refinements:

Hendrickson–Leland algorithm	Walshaw–Cross refinement algorithm
Uses FM algorithm for k-way partitioning. Uses $k(k-1)$ gain tables. Used when k is low.	Almost the same as GKLR. Uses load distribution algorithms for better balancing.

4.3.1 Codebase

The algorithm is implemeted following the paper named **Genetic approaches for graph partitioning: a survey**[5].

- **Language:** C++
- **Graphs & Charts:** C++
- **GitHub Link:** `https://github.com/hyadess/CSE462-Algorithm-Engineering`

4.4 Dataset Description

The dataset used for experiments consists of synthetic graphs generated with varying characteristics to evaluate the performance of the metaheuristic algorithms. The graphs are categorized based on node count and edge probability, which determine the density and structure of the graph.

4.4.1 Graph Types

The dataset includes the following types of graphs:

- **Small Graphs:** Node count ranges from 200 to 800.
- **Large Graphs:** Node count ranges from 3000 to 6000.
- **Dense Graphs:** Edge probability of 0.8, meaning 80% of possible edges exist between vertices.
- **Sparse Graphs:** Edge probability of 0.2, meaning only 20% of possible edges exist between vertices.
- **Random Graphs:** Edge probability of 0.5, representing a medium density graph.

4.4.2 Dataset Creation

The synthetic graphs are generated using the following procedure:

1. For each graph, the number of nodes n is chosen randomly within the specified range (e.g., 200–800 for small graphs).
2. For each pair of nodes u and v, an edge is created with a probability based on the graph type (dense, sparse, or random).
3. Only connected graphs are accepted to ensure meaningful partitions.

4.4.3 Summary of Graph Characteristics

The dataset consists of 100 samples for each of the six graph types, summarized in Table 1.

Graph Type	Node Count	Edge Probability
Small-Dense	200–800	0.8
Small-Sparse	200–800	0.2
Small-Random	200–800	0.5
Large-Dense	3000–6000	0.8
Large-Sparse	3000–6000	0.2
Large-Random	3000–6000	0.5

Table 1: Graph Dataset Characteristics

4.5 Experiments

This section presents the experimental results obtained by applying the metaheuristic algorithms to the described dataset. Each graph evaluates specific parameters such as balance difference, partition count, and algorithm comparisons.

What is Balance Difference

Balance difference defines the allowable deviation in partition sizes during graph partitioning. It is calculated as a percentage of the ideal partition size.

Definition: If the number of vertices is n and the graph is divided into p partitions, each partition ideally contains n/p vertices. A balance difference of $x\%$ allows each partition to deviate up to $n/p \times x/100$ vertices. For instance:

- If $n = 250$, $p = 5$, and $x = 10\%$, each partition can have up to $n/p + n/p \times 0.1 = 55$ vertices.

Impact: Higher balance differences provide more flexibility, potentially reducing the cut size but may result in uneven partition sizes. This trade-off is explored in the following experiments.

4.5.1 Balance Difference vs. Average Cut Size

This experiment investigates how the allowable balance difference affects the average cut size.

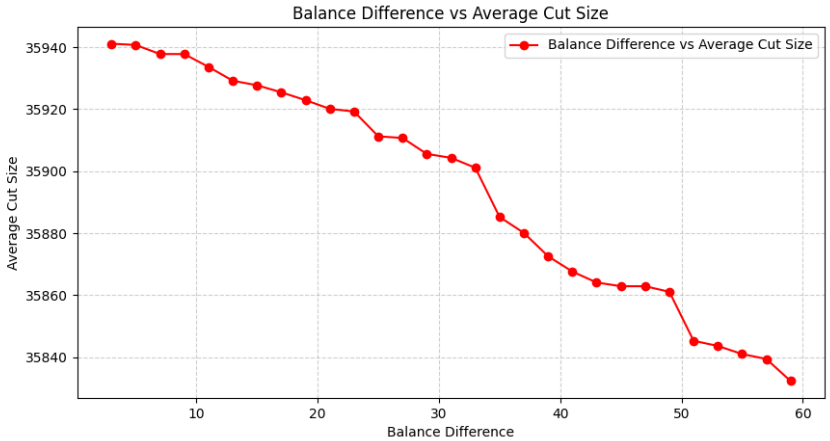

Figure 10: Balance Difference vs. Average Cut Size for Dense Graphs

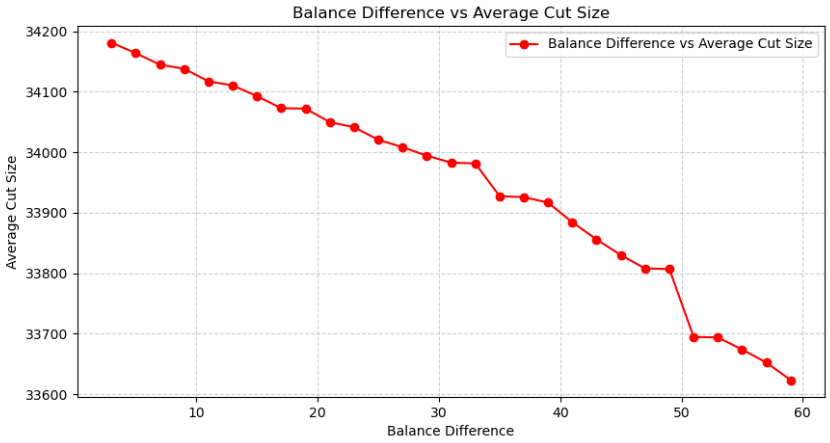

Figure 11: Balance Difference vs. Average Cut Size for Sparse Graphs

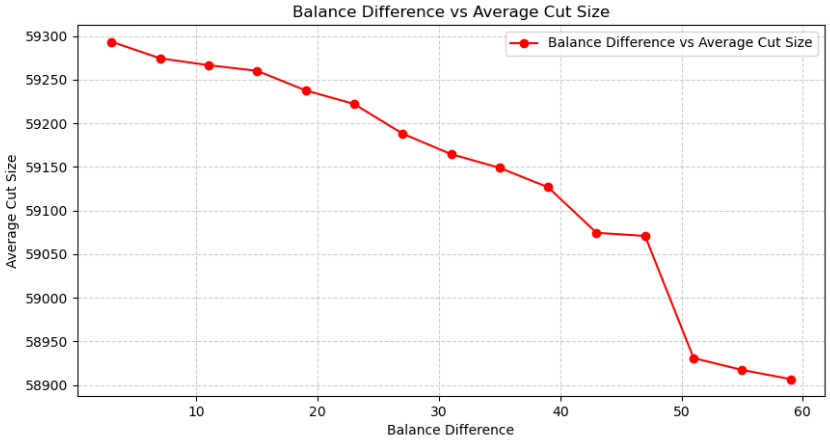

Figure 12: Balance Difference vs. Average Cut Size for Random Graphs

Analysis: Figures 10, 11, and 12 show that increasing the allowable balance difference leads to a decrease in the average cut size. Dense graphs benefit most from greater flexibility, while sparse graphs see marginal improvement.

4.5.2 Partition Count vs. Average Cut Size

The relationship between the number of partitions and the average cut size is explored for three graph types: dense, sparse, and random.

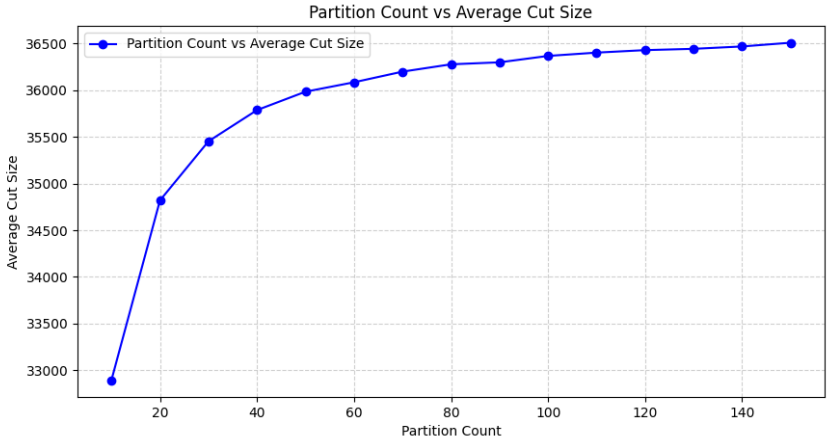

Figure 13: Partition Count vs. Average Cut Size for Dense Graphs

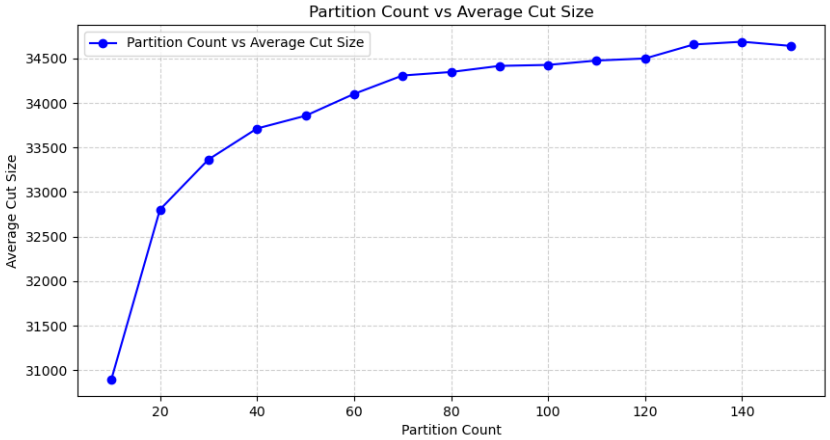

Figure 14: Partition Count vs. Average Cut Size for Sparse Graphs

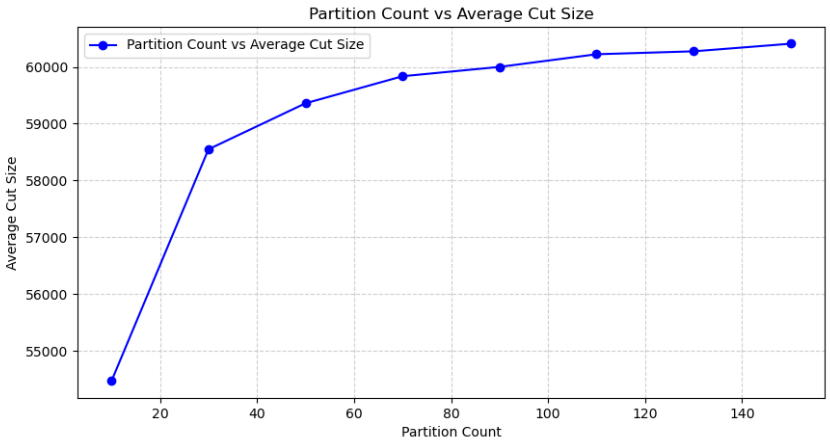

Figure 15: Partition Count vs. Average Cut Size for Random Graphs

Analysis: As seen in Figures 13, 14, and 15, the average cut size increases with the number of partitions. Dense graphs exhibit a linear increase due to their high edge density, while sparse and random graphs demonstrate a slower rise because of their lower connectivity.

4.5.3 Balance Difference vs. Average Shift Count

The effect of balance differences on the average number of shifts required for partitioning is shown below.

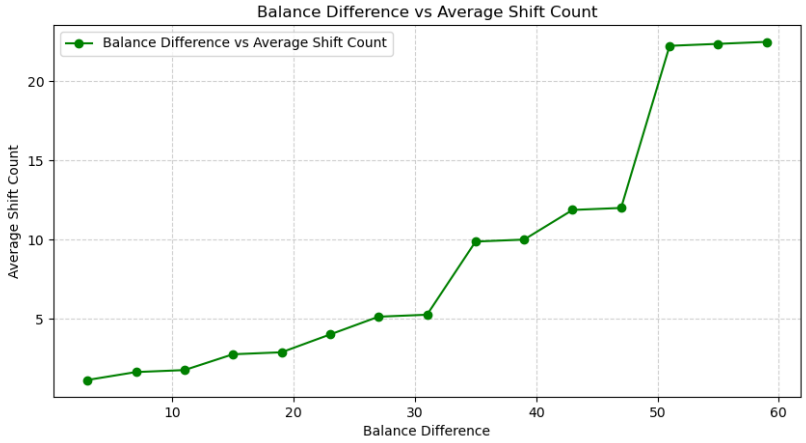

Figure 16: Balance Difference vs. Average Shift Count for Dense Graphs

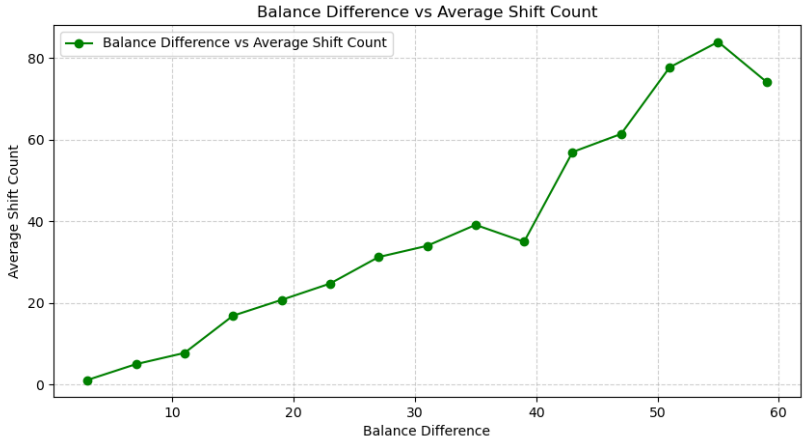

Figure 17: Balance Difference vs. Average Shift Count for Sparse Graphs

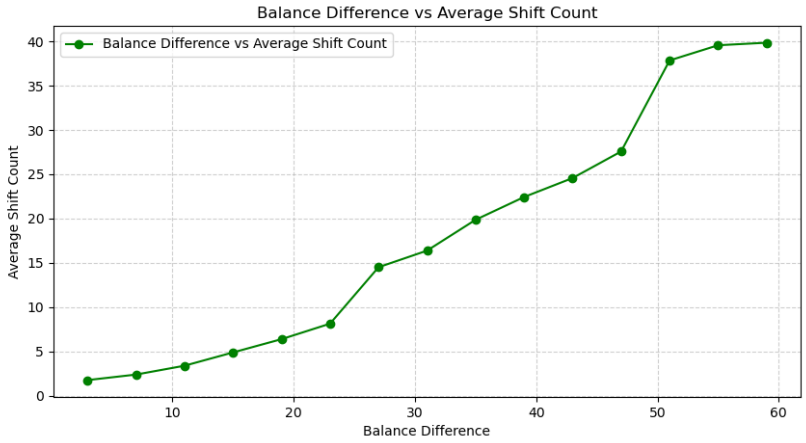

Figure 18: Balance Difference vs. Average Shift Count for Random Graphs

Analysis: Figures 16, 17, and 18 highlight that dense graphs require fewer shifts due to their high connectivity. Sparse and random graphs demand more shifts, indicating greater difficulty in achieving balance. It can be observed better in the Figure19

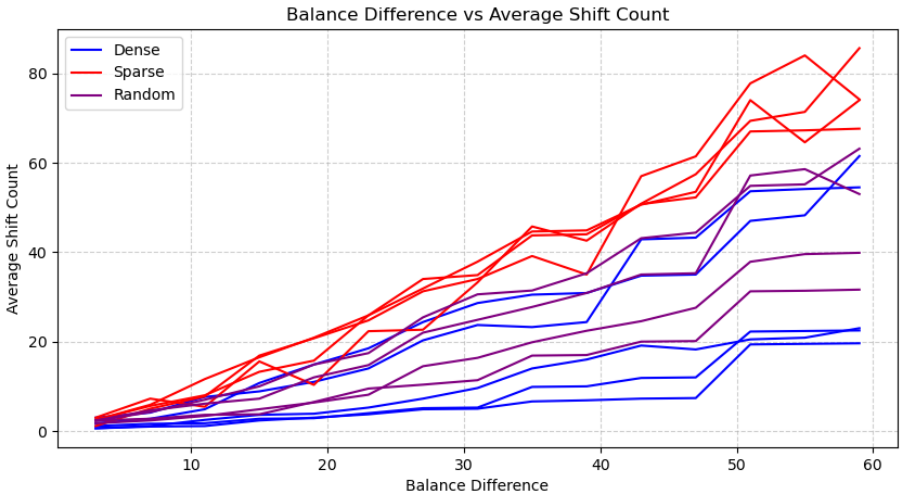

Figure 19: Balance Difference vs. Average Shift Count for all types of Graphs

4.5.4 Performance Comparison using METIS

METIS[4] is a widely used graph partitioning tool known for its efficiency and scalability. It employs multilevel graph partitioning techniques, including coarsening, partitioning, and uncoarsening, to minimize cut size while maintaining balance.

Comparison of the cut size produced by two algorithms across varying partition counts.

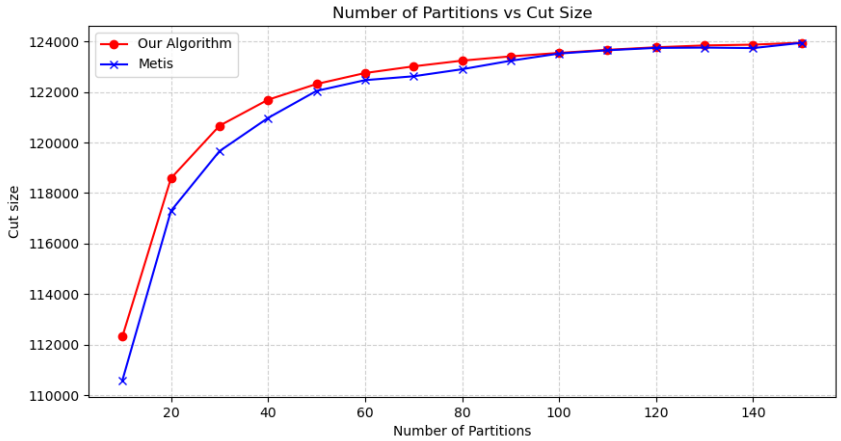

Figure 20: Number of Partitions vs. Cut Size for Dense Graphs (Algorithm Comparison)

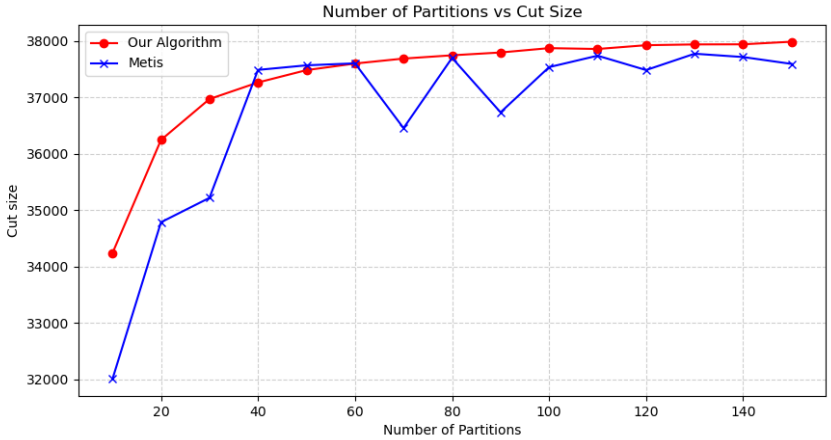

Figure 21: Number of Partitions vs. Cut Size for Sparse Graphs (Algorithm Comparison)

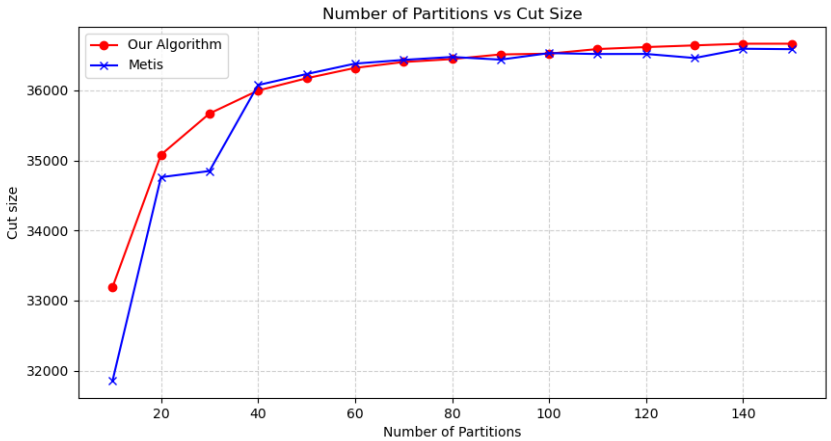

Figure 22: Number of Partitions vs. Cut Size for Random Graphs (Algorithm Comparison)

Analysis: From Figures 20, 21, and 22, METIS generally outperforms our algorithm in dense graphs. Sparse graphs show comparable performance, while random graphs favor our algorithm for higher partition counts. It can be concluded that when the partition count is high, our algorithm performs close to METIS in terms of cut size, but when the partition count is low, METIS performs better.

5 Meta-heuristic Algorithm

Metaheuristic algorithms are high-level procedures designed to explore and exploit large search spaces for complex optimization problems where traditional methods may fail due to the scale or complexity of the solution space. These algorithms, such as Genetic Algorithms (GA), Simulated Annealing (SA), and Particle Swarm Optimization (PSO), do not guarantee an optimal solution but aim to find good approximations within a reasonable computational time. They use mechanisms like randomness, population-based search, and iterative refinement to balance exploration (diversifying the search) and exploitation (intensifying the search near promising areas).

In graph partitioning, metaheuristic algorithms are used to divide a graph into smaller subgraphs while minimizing the number of edges between different partitions (cut edges) and ensuring balanced partition sizes. By representing graph partitions as potential solutions, these algorithms iteratively improve partitions based on fitness criteria such as the number of cut edges and partition balance. Metaheuristics are particularly useful in graph partitioning problems due to their flexibility in handling non-convex, large, and complex search spaces, where exact methods are computationally expensive.

5.1 Genetic Algorithm

5.1.1 Algorithm Explanation

The Genetic Algorithm (GA) is a metaheuristic inspired by the process of natural selection. It iteratively improves a population of candidate solutions by applying bio-inspired operations such as selection, crossover, and mutation. In the context of the graph partitioning problem, each candidate solution represents a possible assignment of nodes to partitions. The fitness of a solution is determined by its ability to minimize the number of cut edges between partitions while maintaining balance in partition sizes.

Input and Initialization

The algorithm takes the following inputs:

- Graph $G = (V, E)$
- Number of partitions k
- Population size p
- Number of generations g
- Mutation probability m
- Weights for cut edges and balance penalties

It starts by initializing a random population of p partitions, where each partition assigns nodes to one of k partitions.

Fitness Calculation

For each partition in the population, calculate the fitness. The fitness is computed based on two factors:

- The number of cut edges (edges crossing partitions)
- The balance between partition sizes

These factors are weighted by c_weight and b_weight, respectively.

Main Evolutionary Loop

For each generation:

- A new population is created by selecting parents, performing crossover, and mutating offspring.
- **Parent Selection**: Two parents are selected from the current population, typically using a method like tournament selection, which selects individuals based on fitness.
- **Crossover**: The two parents are combined to produce a child partition. The crossover mechanism could split the parents' assignments at a random point and combine them to form the child.
- **Mutation**: With probability m, the child is mutated by randomly assigning one node to a different partition. This helps maintain diversity in the population.
- **Fitness Calculation**: The fitness of the child partition is calculated, and the child is added to the new population.
- After all offspring are generated, the current population is replaced by the new population.

Best Partition

At the end of each generation, the algorithm identifies and stores the best partition (the one with the highest fitness).

Output

After g generations, the best partition found during the evolutionary process is returned as the solution.

5.1.2 PseudoCode of GA

The pseudocode of the Genetic Algorithm used for the graph partitioning problem is as follows:

In this algorithm, the fitness function combines the number of cut edges and a balance penalty to assess the quality of partitions. Crossover helps to explore new regions of the solution space by combining two parent solutions, while mutation adds diversity by modifying random parts of a partition.

5.1.3 Time Complexity Analysis

The time complexity of the Genetic Algorithm depends on several factors:

- Let n be the number of nodes in the graph and p be the population size. - Calculating the fitness of a partition involves checking each edge, leading to a fitness evaluation complexity of $O(E)$, where E is the number of edges. - Each generation involves selecting parents, performing crossover, and mutating partitions. Selection takes $O(p)$, crossover and mutation operate on partitions of size n, and fitness calculation is required for each offspring.

Thus, the total time complexity per generation is approximately $O(p \cdot (n+E))$. Given g generations, the overall time complexity is $O(g \cdot p \cdot (n+E))$, where g is the number of generations.

Algorithm 4 Genetic Algorithm for Graph Partitioning

Input: Graph $G = (V, E)$ with n nodes, number of partitions k, population size p, number of generations g, mutation probability m, weight for cut edges c_weight, weight for balance penalty b_weight
Output: Best partition found
Initialize population of p random partitions
for each partition in the population **do**
    Calculate fitness using the fitness function based on cut edges and partition balance
for generation = 1 to g **do** ▷ Loop for each generation
    Create an empty new population
    for each individual in the population **do**
        Select two parents from the current population using a selection method (e.g., tournament selection)
        Perform crossover on the two parents to produce a child partition
        With probability m, mutate the child partition by randomly changing the partition assignment of a node
        Calculate the fitness of the child partition
        Add the child to the new population
    Replace the current population with the new population
    Find and store the best partition in the current population
    Record statistics (best fitness, number of cut edges, partition balance) for this generation
return the best partition found based on fitness

5.1.4 Codebase

The algorithm is implemeted following the paper named **Genetic approaches for graph partitioning: a survey**[5].

- **Language:** Python
- **Graphs & Charts:** Python
- **GitHub Link:** `https://github.com/hyadess/CSE462-Algorithm-Engineering/blob/main/project_codes/genetic/gene.ipynb`

5.1.5 Dataset Description

The Code was run and experimented using the following graphs.

Type	Node	Edge
Sparse	591	34492
Sparse	573	32676
Dense	303	36641
Dense	559	124571
Random	493	60711
Random	384	36839

Table 2: Graph Dataset Description

5.1.6 Experiments

Graph Parameters

In this partitioning problem, the objective is to find an optimal partition based on a fitness function that combines two key factors: the number of cut edges and the balance between partition sizes. The partition sizes, fitness function, and different weights are detailed below.

Partition Sizes: [10, 30, 50, 80, 100]

Fitness Function:

$$\text{Fitness} = \text{CUT\_WEIGHT} \times \text{CUT\_EDGES} + \text{BALANCE\_WEIGHT} \times \text{Balance\_Penalty}$$

Where:

- **CUT_EDGES:** The number of edges that cross between partitions.
- **Balance_Penalty:** The penalty for the imbalance in the size of the partitions.

Weights:

- CUT_WEIGHT = 4, BALANCE_WEIGHT = 1
- CUT_WEIGHT = 1, BALANCE_WEIGHT = 3
- CUT_WEIGHT = 1, BALANCE_WEIGHT = 1

These parameters define the trade-off between minimizing the number of cut edges and maintaining balanced partitions.

Avarage Cut Size and Average Size Difference

Sparse Graph:

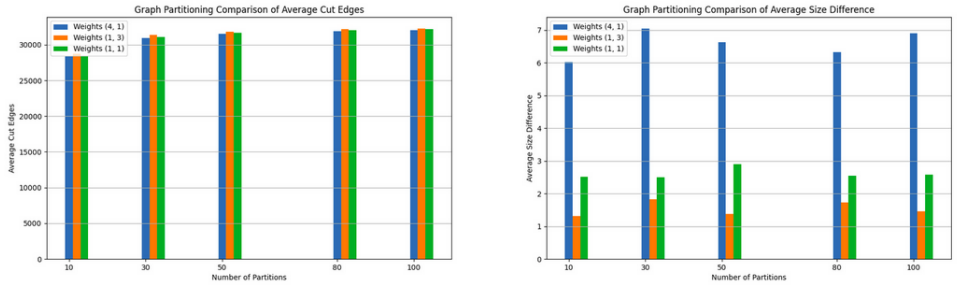

Figure 23: Graph Partitioning Comparison of Average Cut Edges and Average Size Difference for Sparse Graph

- **Graph on the left (Average Cut Edges):** The number of cut edges is very similar across all partition sizes and for all three weight combinations. This suggests that changing the weights has little impact on the number of edges cut between partitions.
- **Graph on the right (Average Size Difference):** The size difference between partitions is highest for the weight combination of $(4, 1)$, indicating that prioritizing cut edges (higher weight on cut edges) leads to larger imbalances in partition sizes. As the balance weight increases (combinations of $(1, 3)$ and $(1, 1)$), the size difference decreases, resulting in more balanced partitions.

In summary, minimizing cut edges leads to higher size imbalances, and increasing the balance weight improves partition size balance without significantly affecting the number of cut edges.

Random Graph:

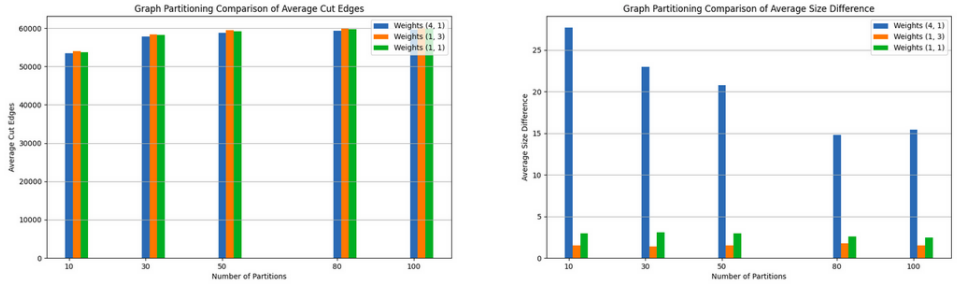

Figure 24: Graph Partitioning Comparison of Average Cut Edges and Average Size Difference for Random Graph

- **Graph on the left (Average Cut Edges):** The number of cut edges is fairly consistent across all partition sizes and weight configurations, indicating that the choice of weights has minimal impact on reducing the number of edges cut between partitions. However, there is a slight reduction in cut edges for higher partition sizes.
- **Graph on the right (Average Size Difference):** A higher weight on the cut edges (Weight $(4, 1)$) leads to significantly greater size imbalances, particularly when the number of partitions is small. As the balance weight increases (Weight $(1, 3)$ and $(1, 1)$), the size difference becomes smaller, indicating more balanced partitions. The $(1, 1)$ weight combination achieves the smallest size difference across all partition sizes.

In summary, higher emphasis on cut edges results in worse partition balance, whereas increasing the balance weight leads to more equitable partitioning without drastically affecting the number of cut edges.

Dense Graph :

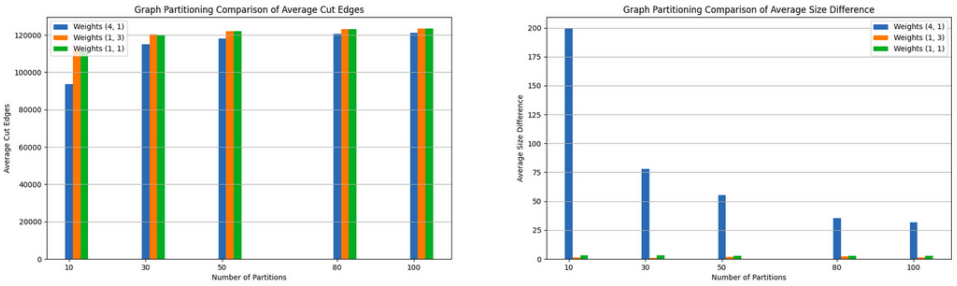

Figure 25: Graph Partitioning Comparison of Average Cut Edges and Average Size Difference for Dense Graph

- **Graph on the left (Average Cut Edges):** The number of cut edges increases with the number of partitions across all weight configurations. Weight configuration $(4, 1)$ has a slight advantage in minimizing cut edges for higher partition sizes, while weights $(1, 3)$ and $(1, 1)$ perform similarly. The difference is most notable for smaller partition sizes.
- **Graph on the right (Average Size Difference):** As expected, when more emphasis is placed on minimizing cut edges (Weights $(4, 1)$), the size imbalance increases significantly. The average size difference is much larger with $(4, 1)$ weights, especially when the number of partitions is small. As the balance weight increases (Weights $(1, 3)$ and $(1, 1)$), the size differences decrease dramatically, indicating that these configurations lead to more balanced partitions across all partition sizes.

In conclusion, configurations with higher cut weight (such as $(4, 1)$) result in lower cut edges but larger imbalances in partition sizes. Conversely, configurations with higher balance weight (such as $(1, 3)$ and $(1, 1)$) achieve more balanced partitions at the expense of slightly increased cut edges.

Cut Edge vs Generation and Difference Size vs Generation

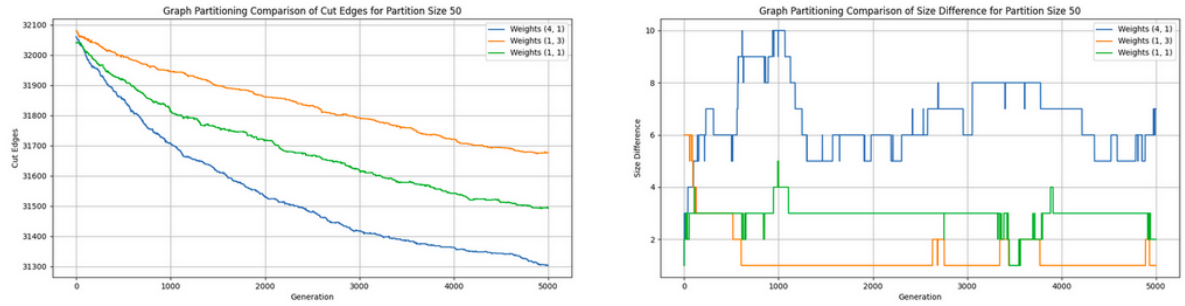

Figure 26: Graph partitioning comparison for partition size 50 in Sparse Graph.

The figure shows a comparison of graph partitioning performance for partition size 50 across different weight configurations. The left plot displays the reduction in the number of cut edges over generations, where configurations with higher weight ratios (e.g., (4, 1)) converge more slowly. The right plot illustrates the size difference between partitions, with balanced weights (e.g., (1, 1)) showing greater stability and lower variability.

The Random and Dense graphes show similar results.

Comparision :

Comparision on Number of Cut Edges :

Type	Node	Edge
Sparse	573	32676
Random	493	60711
Dense	559	124571

Partition Count -> 20

Maximum Partition Size Difference -> 1

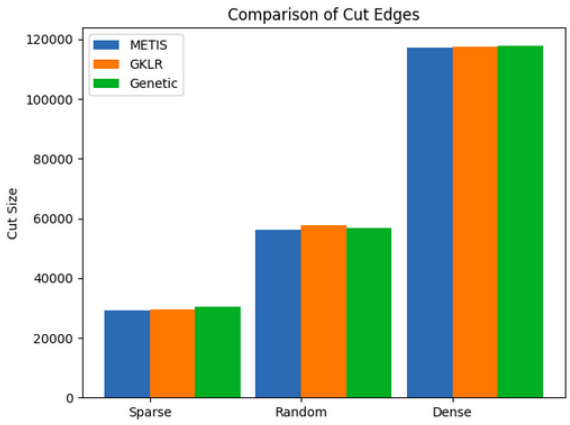

Figure 27: METIS vs GKLR vs Genetic

Accuracy :

Type	METIS	GKLR	Genetic
Sparse	29269	29635	30578
Random	56172	57738	56802
Dense	117296	117689	117981

$$\frac{GKLR}{METIS} = 1.0115$$

$$\frac{Genetic}{METIS} = 1.013$$

$$\frac{Genetic}{GKLR} = 1.0015$$

Figure 28: Accuracy

5.2 Some Other Metaheuristic Algorithms

5.2.1 Simulated Annealing

Simulated Annealing (SA) is a probabilistic optimization technique inspired by the annealing process in metallurgy. When applied to graph partitioning, it aims to minimize the number of edges between partitions (cut edges) while keeping the sizes of the partitions balanced. The algorithm explores different partitions by making small modifications to an initial solution and probabilistically accepts worse solutions in the hope of avoiding local minima. As the algorithm progresses, it gradually reduces the probability of accepting worse solutions, allowing it to converge to a near-optimal solution. The algorithm was used in a paper [7].

5.2.2 Discrete Particle Swarm Optimization

In Discrete Particle Swarm Optimization (DPSO), particles represent potential solutions for optimization problems, with positions encoded as binary vectors. Particles move through the search space by updating their velocities and positions based on personal and global best solutions. The movement is determined

Algorithm 5 Simulated Annealing for Graph Partitioning

Input: Graph $G = (V, E)$, number of partitions k, initial temperature T_0, cooling rate α, number of iterations n
Output: Best partition found
Initialize partition P randomly
Calculate the initial cost C_P of partition P
Set $T \leftarrow T_0$
Set $P_{\text{best}} \leftarrow P$ and $C_{\text{best}} \leftarrow C_P$
for $i = 1$ to n **do**
    Generate a neighboring partition P_{new} by moving or swapping nodes
    Calculate the cost C_{new} of P_{new}
    if $C_{\text{new}} < C_P$ **then**
        Accept the new partition: $P \leftarrow P_{\text{new}}, C_P \leftarrow C_{\text{new}}$
    else
        Compute acceptance probability $p = \exp\left(\frac{-(C_{\text{new}} - C_P)}{T}\right)$
        if Random number $r \in [0, 1] < p$ **then**
            Accept the new partition: $P \leftarrow P_{\text{new}}, C_P \leftarrow C_{\text{new}}$
    if $C_P < C_{\text{best}}$ **then**
        Update the best partition: $P_{\text{best}} \leftarrow P, C_{\text{best}} \leftarrow C_P$
    Cool the system: $T \leftarrow \alpha T$
return P_{best}

by a velocity update equation, influenced by inertia, personal experience, and collective experience. The position update is handled using a probabilistic function, often a sigmoid. The process continues iteratively until a termination criterion is met.

DPSO is used in a paper[1] for graph partitioning.

6 Exact and Approximation Algorithm

6.1 Exact Algorithms

An **exact algorithm** is a method that uses mathematical techniques to find the best solution to an optimization problem, or to prove that no solution exists.

6.1.1 Branch and Bound

Branch and Bound is an exact algorithm for solving combinatorial optimization problems, where the solution space is systematically divided (branching), and lower or upper bounds are computed to eliminate parts of the space (bounding). The algorithm [6] explores the space recursively to find the optimal partition.

Definition: The Branch and Bound algorithm systematically searches through all possible partitions of the graph, using bounds to prune unpromising solutions and reduce computational complexity.

6.1.2 Integer Linear Programming (ILP)

Integer Linear Programming formulates the graph partitioning problem as a set of linear constraints and objectives. It seeks to minimize the number of edges between partitions while balancing partition sizes. The algorithm is taken from a paper [3].

Algorithm 6 Discrete Particle Swarm Optimization (DPSO)

Input: Number of particles n, inertia weight w, learning factors c_1, c_2, maximum iterations T
Output: Best solution g_{best}
Initialize positions x_i and velocities v_i of particles
Initialize personal best $p_{\text{best},i}$ for each particle
Initialize global best g_{best}
for iteration $t = 1$ to T **do**
    for each particle i **do**
        Update velocity:

$$v_i^{t+1} = w \cdot v_i^t + c_1 \cdot r_1 \cdot (p_{\text{best},i} - x_i^t) + c_2 \cdot r_2 \cdot (g_{\text{best}} - x_i^t)$$

        Update position:

$$x_i^{t+1} = \begin{cases} 1, & \text{if } \sigma(v_i^{t+1}) > r \\ 0, & \text{otherwise} \end{cases}$$

        Evaluate fitness of new position x_i^{t+1}
        if fitness of x_i^{t+1} is better than $p_{\text{best},i}$ **then**
            Update $p_{\text{best},i} \leftarrow x_i^{t+1}$
    if any $p_{\text{best},i}$ is better than g_{best} **then**
        Update $g_{\text{best}} \leftarrow p_{\text{best},i}$
    if termination criteria met (e.g., maximum iterations) **then**
        Exit loop
return g_{best}

Algorithm 7 Branch and Bound for Graph Partitioning

Input: Graph $G = (V, E)$, number of partitions k
Output: Optimal partition of G
Initialize empty partition P
Define a global bound $B = \infty$
Procedure: Recursive Branch and Bound
Partition V into k sets (branching)
Calculate lower bound on partition quality (bounding)
if bound $< B$ **then**
    Explore the branch
    if valid partition **then**
        Update B and store the partition
else
    Prune the branch
Return the partition with the best bound

Algorithm 8 ILP for Graph Partitioning

Input: Graph $G = (V, E)$, number of partitions k
Output: Optimal partition of G
Define binary variables $x_{i,j}$ where $x_{i,j} = 1$ if node i is in partition j, otherwise 0
Define objective function to minimize cut edges: $\sum_{(i,j)\in E} |x_{i,p} - x_{j,q}|$
Add constraints:
    Each node i is in exactly one partition: $\sum_j x_{i,j} = 1$
    Partition size constraints to balance partitions
Solve using an ILP solver
Return optimal partition

Definition: ILP formulates graph partitioning as an optimization problem with linear constraints and objectives. The solution is computed using ILP solvers, making it exact but often computationally expensive.

6.1.3 Randomized Greedy

Randomized Greedy is a heuristic algorithm that iteratively assigns nodes to partitions based on a random order. It greedily selects the best partition for each node, ensuring a valid solution is obtained.

Algorithm 9 Randomized Greedy for Graph Partitioning

Input: Graph $G = (V, E)$, number of partitions k
Output: Valid partition of G
Shuffle the node set V
for each node v in shuffled V **do**
    Calculate the cost of placing v in each partition
    Place v in the partition with the minimal cost
Return the final partition

Definition: Randomized Greedy is an exact algorithm that shuffles the order of nodes and places them in partitions based on greedy optimization, ensuring balanced and minimal cut partitions.

6.2 Approximation Algorithms

Approximation algorithms are polynomial time algorithms that produce approximate solutions to NP-hard optimization problems, with demonstrable guarantees on the quality of the solution.

6.2.1 Spectral Partitioning

Spectral Partitioning is an approximation algorithm that uses eigenvalues and eigenvectors of the Laplacian matrix of the graph to partition the nodes. The nodes are divided based on the signs of the components of the second smallest eigenvector. The algorithm was used in paper [2]

Definition: Spectral Partitioning approximates graph partitioning by leveraging the spectral properties (eigenvalues and eigenvectors) of the graph's Laplacian matrix. It is efficient but not guaranteed to provide an optimal solution.

7 Application

Graph partitioning has a wide range of real-life applications across various fields. Some notable examples are:

Algorithm 10 Spectral Partitioning for Graph Partitioning

1: **Input:** Graph $G = (V, E)$, number of partitions k
2: **Output:** Approximate partition of G
3: Compute the Laplacian matrix L of graph G
4: Compute the eigenvalues and eigenvectors of L
5: Sort nodes based on the values of the second smallest eigenvector
6: Partition nodes based on the sign of the eigenvector components
7: Return the partition

- **Parallel Computing:** In high-performance computing, graph partitioning is used to divide large graphs into smaller subgraphs that can be processed in parallel, improving computational efficiency.
- **VLSI Design:** In Very-Large-Scale Integration (VLSI) design, graph partitioning helps in dividing the circuit layout into smaller blocks, optimizing the placement and routing of components on a chip.
- **Load Balancing in Distributed Systems**: Distributes workload evenly across multiple computing nodes while minimizing the communication overhead between nodes. The system's tasks and their dependencies are modeled as a graph, where balanced partitions ensure efficient resource utilization.
- **Social Network Analysis:** Graph partitioning is used to identify communities or clusters within social networks, helping in understanding the structure and dynamics of social interactions.
- **Biology and Medicine:** In bioinformatics, graph partitioning is applied to analyze biological networks, such as protein-protein interaction networks, to identify functional modules or pathways.
- **Telecommunications:** Graph partitioning is used in network design and optimization, such as dividing a network into subnetworks for efficient routing and load balancing.
- **Traffic Management:** In transportation and logistics, graph partitioning helps in optimizing traffic flow, route planning, and resource allocation.
- **Data Mining and Big Data:** Graph partitioning is used to manage and analyze large datasets by dividing them into manageable chunks, enabling faster processing and analysis.
- **Circuit Design:** In electrical engineering, graph partitioning is used to design and optimize circuits, ensuring efficient use of resources and minimizing power consumption.
- **Image Analysis:** It helps in segmenting images into meaningful regions, improving object recognition and image processing tasks.
- **Smart City Design:** Facilitates efficient resource allocation, traffic management, and urban planning by dividing the city network into manageable zones.
- **Wireless Communications:** Enhances network design and load balancing, ensuring optimal use of bandwidth and minimizing interference.
- **Data Analysis:** Makes handling large datasets more efficient by breaking them into smaller, easier-to-analyze parts, speeding up the processing and gaining insights.

References

[1] Naresh Ghorpade and HR Bhapkar. "Novel multilevel particle swarm optimization algorithm for graph partitioning". In: *J. Math. Comput. Sci.* 12 (2022), Article–ID.

[2] Stephen Guattery and Gary L Miller. "On the performance of spectral graph partitioning methods". In: *SODA*. Vol. 95. Citeseer. 1995, pp. 233–242.

[3] Alexandra Henzinger, Alexander Noe, and Christian Schulz. "ILP-based local search for graph partitioning". In: *Journal of Experimental Algorithmics (JEA)* 25 (2020), pp. 1–26.

[4] George Karypis and Vipin Kumar. *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices*. Sept. 1998.

[5] Jin Kim et al. "Genetic approaches for graph partitioning: a survey". In: *Proceedings of the 13th annual conference on Genetic and evolutionary computation*. 2011, pp. 473–480.

[6] Jenny Nossack and Erwin Pesch. "A branch-and-bound algorithm for the acyclic partitioning problem". In: *Computers & operations research* 41 (2014), pp. 174–184.

[7] Lixin Tao et al. "Simulated annealing and tabu search algorithms for multiway graph partition". In: *Journal of Circuits, Systems, and Computers* 2.02 (1992), pp. 159–185.