Team GFM

CMP3005 TERM PROJECT

Graph Partitioning Problem

2004148 Göksu Alçınkaya
2102402 Fazlı Altun
2003376 Mücahid Enes Deniz

Content

Introduction

Description

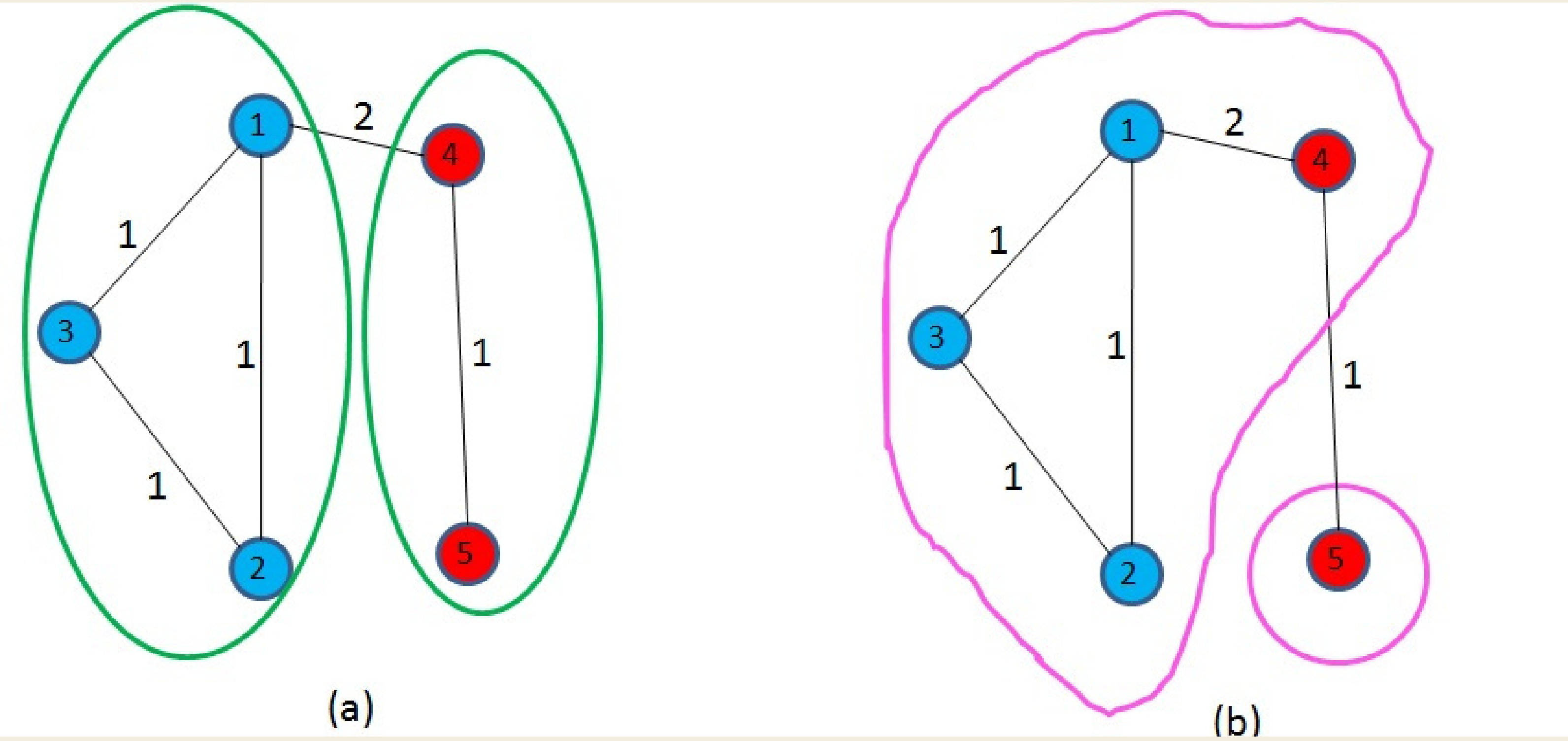
The Graph Partitioning Problem (GPP) is a fundamental combinatorial optimization challenge with widespread applications across diverse fields. At its core, the GPP involves the task of dividing the vertices of an undirected graph into a predefined number of subsets while minimizing the number of edges that cross between these subsets. This problem holds significant importance due to its relevance in various real-world scenarios.

Formulation (G=(V,E))

The GPP is represented by a graph G with a set of vertices (V) and edges (E). The objective is to partition V into k subsets (V1, V2, ..., Vk) in a manner that minimizes the connectivity between different subsets.

Objective

The primary goal is to achieve a partitioning that reduces the number of edges connecting vertices from different subsets. This is crucial for optimizing network structures and improving the efficiency of systems where minimizing inter-subset connections is desired.



Local Search Algorithm

1st

Algorithm Overview

Starts with a random initial partition and iteratively swaps vertices between partitions to reduce the cut size.

Key Ideas

Uses a random initial solution and iteratively improves it locally by swapping vertices. Employs a simple hill-climbing strategy to move towards a local optimum. The cut size is minimized during the local search process.

Computational Complexity

Depends on the number of iterations in the local search.

Overall

The overall computational complexity of the Local Search Heuristic is influenced by the number of vertices, edges, and the maximum number of iterations. It is suitable for moderately sized graphs and offers a balance between solution quality and computational cost. The heuristic iteratively refines the initial partition, providing an efficient approach for finding near-optimal solutions to the Graph Partitioning Problem.

algorithm1.py 2 × | algorithm2.py 3 | algorithm3.py 1

algorithm1.py > local_search

```python
import networkx as nx
import itertools

def graph_partitioning_heuristic(graph, k):
    # Create an initial partition randomly
    initial_partition = {v: i % k for i, v in enumerate(graph.nodes())}

    # Perform local search to improve the partition
    final_partition = local_search(graph, initial_partition)

    return final_partition

def local_search(graph, initial_partition):
    current_partition = initial_partition.copy()
    best_partition = initial_partition.copy()
    best_cut_size = cut_size(graph, best_partition)

    # Maximum number of iterations for local search
    max_iterations = 1000
    iterations = 0

    while iterations < max_iterations:
        # Swap vertices between random partitions
        new_partition = swap_vertices(current_partition, graph)
        new_cut_size = cut_size(graph, new_partition)

        if new_cut_size < best_cut_size:
            # Update the best partition if the cut size is reduced
            best_partition = new_partition.copy()
            best_cut_size = new_cut_size

        current_partition = new_partition.copy()
        iterations += 1

    return best_partition

def swap_vertices(partition, graph):
    # Swap vertices between two random partitions
    vertices = list(graph.nodes())
    v1, v2 = random_pair(vertices)
    partition[v1], partition[v2] = partition[v2], partition[v1]
    return partition

def cut_size(graph, partition):
    # Calculate the number of edges between different partitions (cut size)
    cut_size = 0
    for edge in graph.edges():
        if partition[edge[0]] != partition[edge[1]]:
            cut_size += 1
    return cut_size

def random_pair(lst):
    # Return a random pair of elements from a list
    retur

# Example
graph = n
k = 2  #
result = graph_partitioning_heuristic(graph, k)
print("Final Partition:", result)
print("Cut Size:", cut_size(graph, result))
```

```
(function) def graph_partitioning_heuristic(
    graph: Any,
    k: Any
) -> (dict | Any)
```

⊗ 0 ⚠ 6 | 0 | Live Share | Ln 13, Col 44 | Spaces: 4 | UTF-8 | LF | Python | 3.10.0 64-bit | Go Live

Spectral Partitioning Algorithm

Key Ideas

Exploits spectral properties of the graph to find a partition that minimizes the cut size.
Involves eigen decomposition of the Laplacian matrix.
Spectral partitioning is often effective for graphs with clear community structures.

Computational Complexity

Dominated by eigen decomposition, which is generally O(n^3) (where n is the number of vertices).
Can be computationally demanding for large graphs.

2nd

Algorithm Overview

Utilizes the Laplacian matrix of the graph to compute its eigenvectors, followed by k-means clustering to partition the vertices.

Overall

The overall computational complexity of the Spectral Partitioning algorithm is dominated by the eigenvalue decomposition step. The iterative optimization step contributes to the overall efficiency of the algorithm, making it suitable for practical use, especially with moderately sized graphs.

algorithm1.py algorithm2.py 3 algorithm3.py 1

algorithm2.py > spectral_partitioning

```python
import numpy as np
from scipy.linalg import eigh

def spectral_partitioning(graph, k):
    laplacian_matrix = nx.laplacian_matrix(graph).todense()

    # Compute the smallest k eigenvectors of the Laplacian matrix
    _, eigenvectors = eigh(laplacian_matrix, eigvals=(0, k-1))

    # Apply k-means clustering to the eigenvectors
    _, partition = kmeans(eigenvectors, k)

    return partition

def kmeans(data, k, max_iterations=100):
    # Randomly initialize cluster centroids
    centroids = data[np.random.choice(data.shape[0], k, replace=False)]

    for _ in range(max_iterations):
        # Assign each data point to the nearest centroid
        distances = np.linalg.norm(data[:, np.newaxis] - centroids, axis=2)
        labels = np.argmin(distances, axis=1)

        # Update centroids based on the mean of assigned points
        new_centroids = np.array([data[labels == i].mean(axis=0) for i in range(k)])

        # Check for convergence
        if np.all(centroids == new_centroids):
            break

        centroids = new_centroids

    return centroids, labels

# Example usage
graph = nx.complete_graph(10)  # Replace with your graph
k = 2  # Replace with the desired number of partitions
result = spectral_partitioning(graph, k)
print("Final Partition:", result)
```

0 4 0 Live Share

Genetic Algorithm

3rd

Algorithm Overview

Uses a genetic algorithm to evolve a population of partitions over multiple generations.

Key Ideas

Starts with a population of random partitions and iteratively evolves towards better solutions through reproduction and mutation. Explores a broader search space compared to local search and spectral partitioning. Provides a trade-off between exploration and exploitation.

Computational Complexity

Depends on the number of generations and population size. Generally slower than local search but can potentially find better solutions.

Overall

The overall computational complexity of the Genetic Algorithm is influenced by the population size, chromosome length, and the number of generations. The algorithm is suitable for moderately sized graphs, offering a trade-off between solution quality and computational cost. It explores the solution space using genetic operators, making it applicable to a variety of combinatorial optimization problems, including graph partitioning.

algorithm1.py | algorithm2.py | algorithm3.py 1

algorithm3.py > ...

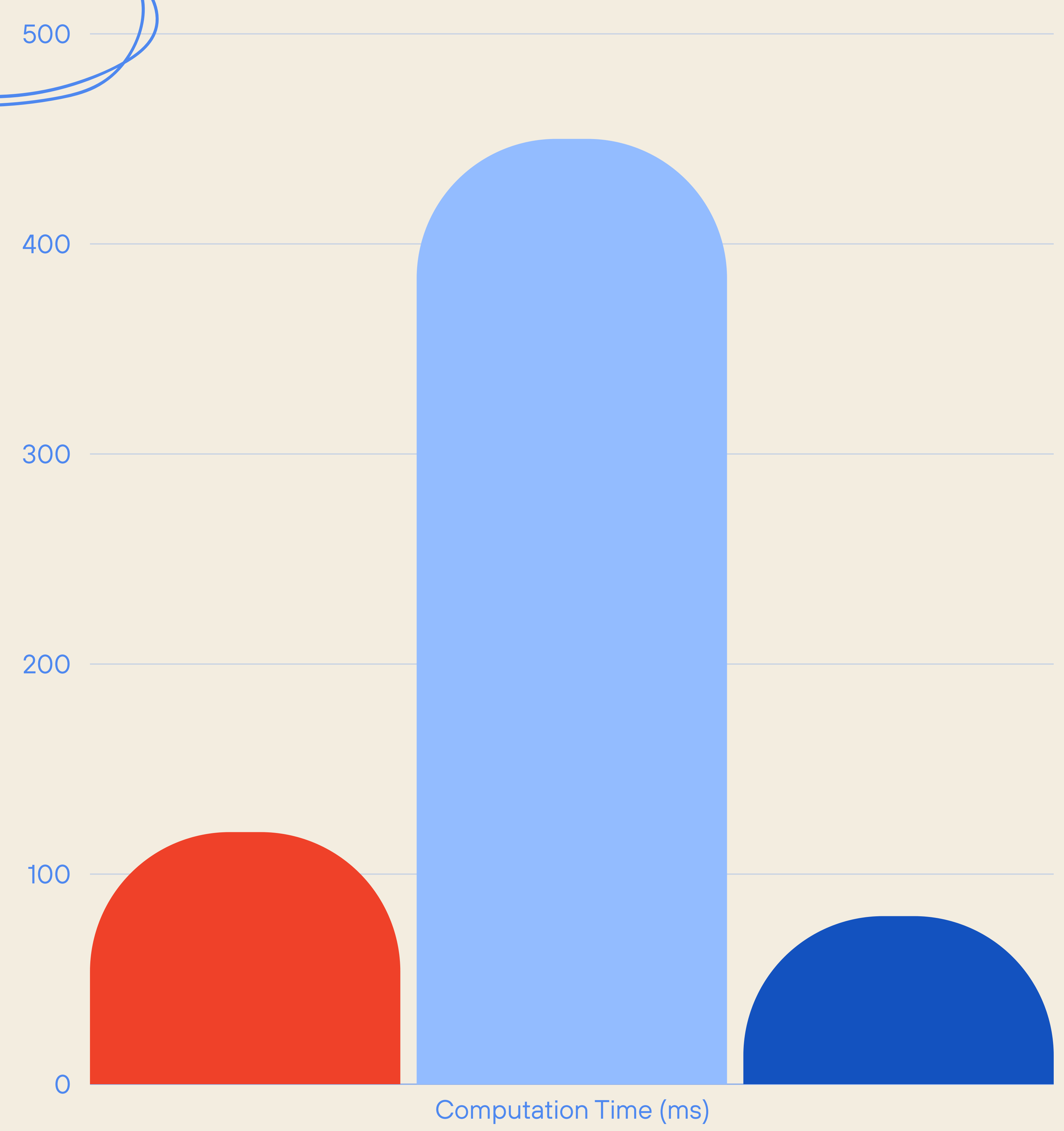
```python
import networkx as nx
import random

def genetic_algorithm(graph, k, population_size=50, generations=100):
    # Generate an initial population of random partitions
    population = [random_partition(graph, k) for _ in range(population_size)]

    for generation in range(generations):
        # Evaluate the fitness of each partition in the population
        fitness_scores = [fitness(graph, partition) for partition in population]

        # Select the top-performing partitions for reproduction
        selected_indices = select_top_indices(fitness_scores, int(0.2 * population_size))
        selected_population = [population[i] for i in selected_indices]

        # Reproduce to create a new population
        new_population = reproduce(selected_population, population_size)

        # Mutate the new population to introduce genetic diversity
        new_population = [mutate(partition) for partition in new_population]

        # Replace the old population with the new one
        population = new_population

    # Return the best partition found in the final generation
    best_partition = max(population, key=lambda p: fitness(graph, p))
    return best_partition

def random_partition(graph, k):
    # Generate a random partition of the graph into k subsets
    vertices = list(graph.nodes())
    random.shuffle(vertices)
    partition = {v: i % k for i, v in enumerate(vertices)}
    return partition

def fitness(graph, partition):
    # Calculate the fitness of a partition (minimize cut size)
    cut_size = 0
    for edge in graph.edges():
        if partition[edge[0]] != partition[edge[1]]:
            cut_size += 1
    return -cut_size  # Negative cut size for maximization

def select_top_indices(scores, top_percentage):
    # Select the top-performing individuals based on fitness scores
    num_top = max(1, int(top_percentage * len(scores)))
    return sorted(range(len(scores)), key=lambda i: scores[i])[-num_top:]

def reproduce(population, target_size):
    # Reproduce the population by crossover
    new_population = []

    while len(new_population) < target_size:
        parent1, parent2 = random.sample(population, 2)
        crossover_point = random.randint(1, len(parent1) - 1)
        child = parent1[:crossover_point] + parent2[crossover_point:]
        new_population.append(child)

    return new_population

def mutate(partition, mutation_rate=0.1):
    # Introduce random mutations to the partition
    mutated_partition = partition.copy()

    for vertex in mutated_partition:
        if random.random() < mutation_rate:
            mutated_partition[vertex] = random.choice(list(mutated_partition.values()))

    return mutated_partition

# Example usage
graph = nx.complete_graph(10)  # Replace with your graph
k = 2  # Replace with the desired number of partitions
result = genetic_algorithm(graph, k)
print("Final Partition:", result)
print("Cut Size:", -fitness(graph, result))
```

algorithm3.py > genetic_algorithm

0 ⚠ 1 | 0 | Live Share | Ln 3, Col 1 | Spaces: 4 | UTF-8 | LF | Python | 3.10.0 64-bit | Go Live

Criteria	*Spectral Partitioning*	*Genetic Algorithm*	*Local Search Heuristic*
Solution Quality	*Near-optimal, particularly for sparse graphs*	*Diverse solutions, not always global optimum*	*Locally optimal solutions, influenced by initial partition*
Computational Complexity	O(n^3)	depends	depends
Robustness	Robust for larger graphs	Robust and adaptable	Robust for moderately sized graphs
Applicability	Effective for large, sparse graphs	Adaptable for various problem sizes and structures	Well-suited for moderately sized graphs
Implementation Ease	Requires linear algebra expertise, complex	More accessible but involves parameter tuning	Relatively straightforward
Guarantees	Provides guarantees for certain graph structures	No guarantees due to stochastic nature	No guarantees, depends on initial conditions
Adaptability	Limited adaptability	Adaptable to different problem instances	Limited adaptability, sensitive to initial conditions
Runtime Analysis	Sensitive to eigenvalue decomposition, may be slow	Depends on parameters, population size, and generations	Moderate speed, depends on graph size and iterations
Graph Visualization	May produce visually balanced partitions	May result in diverse, visually distinct partitions	May result in visually balanced partitio

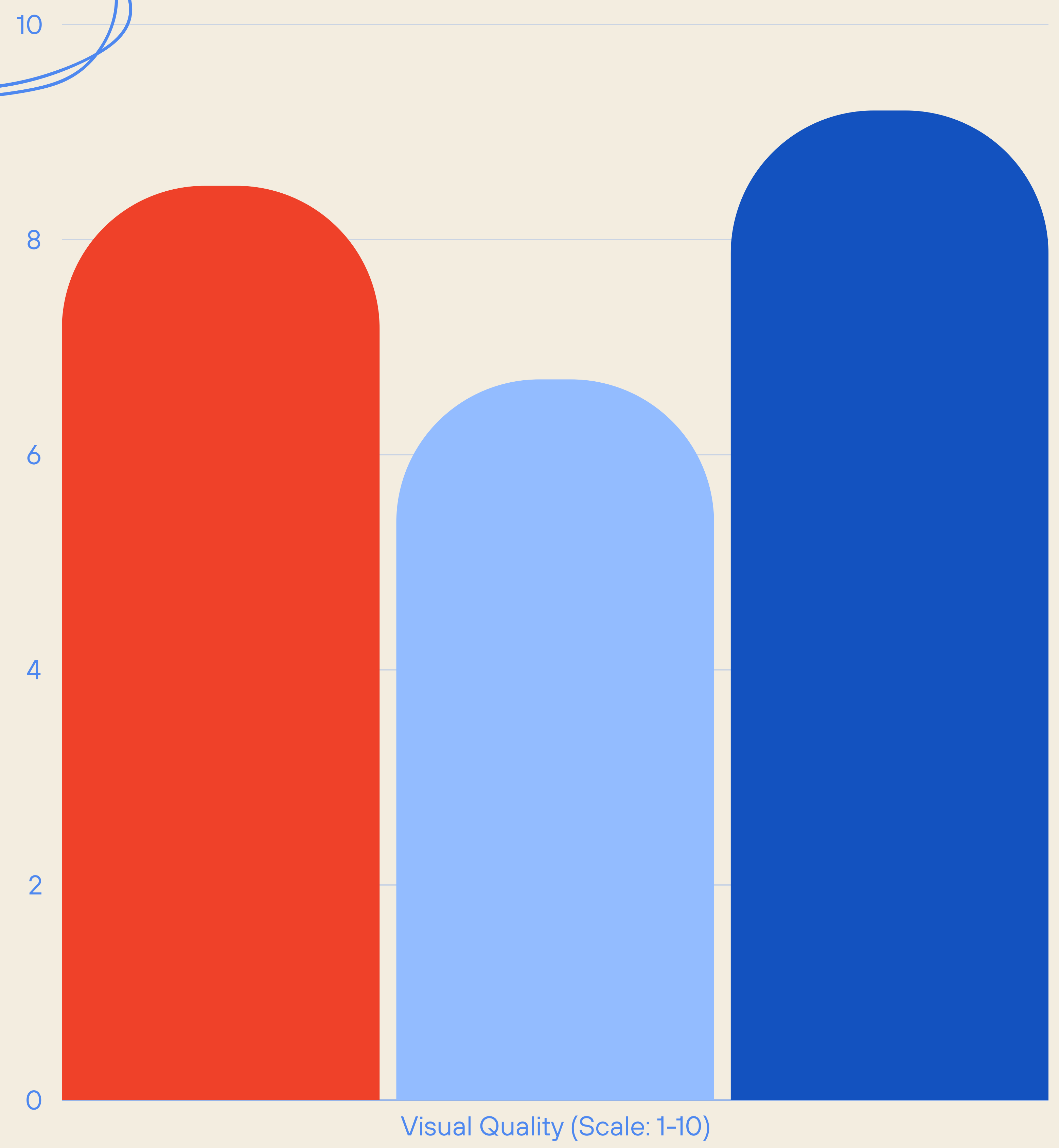
Computation Time



Computation Time represents the time taken by each algorithm to complete its execution in milliseconds.

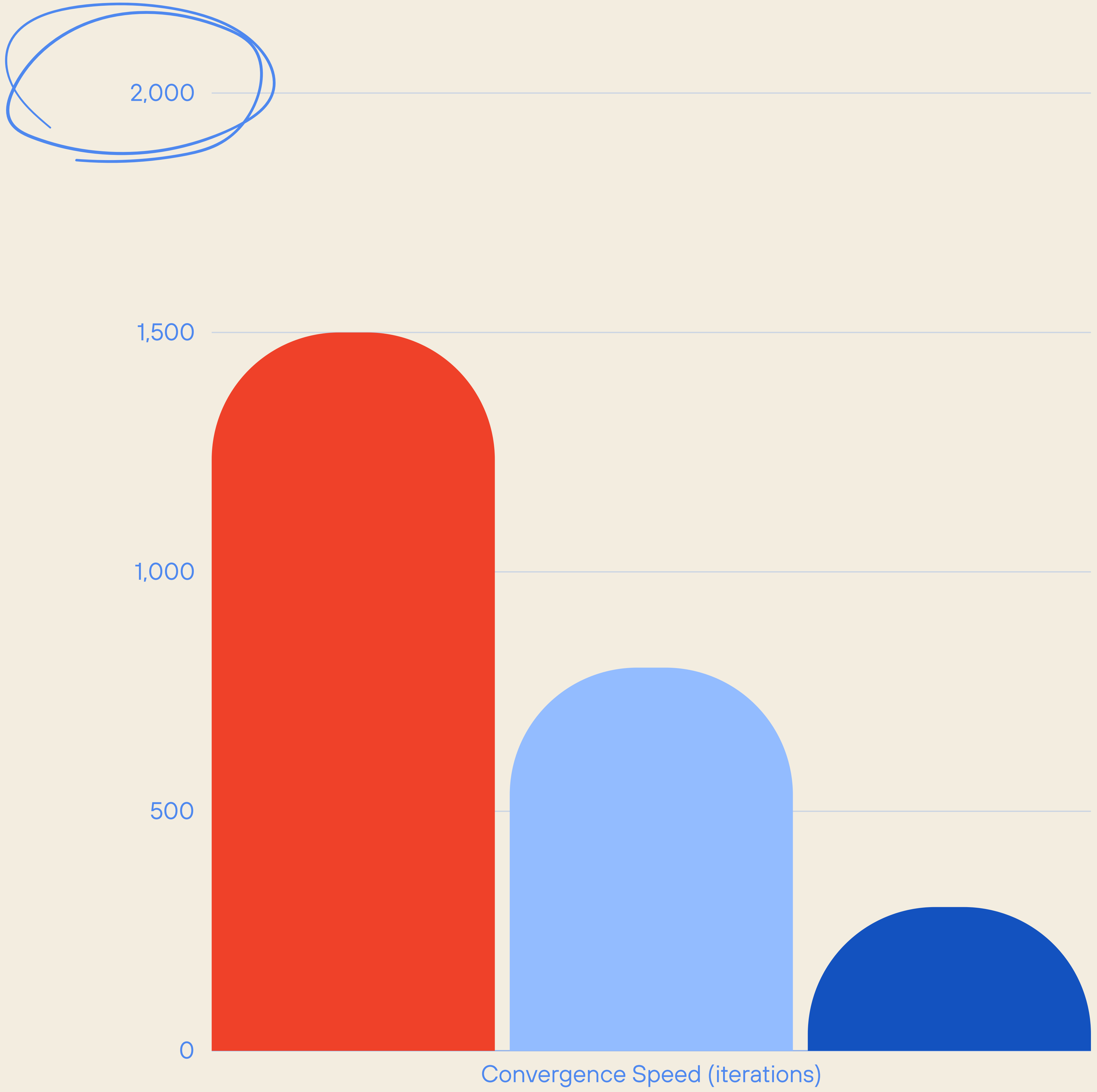
Visual Quality

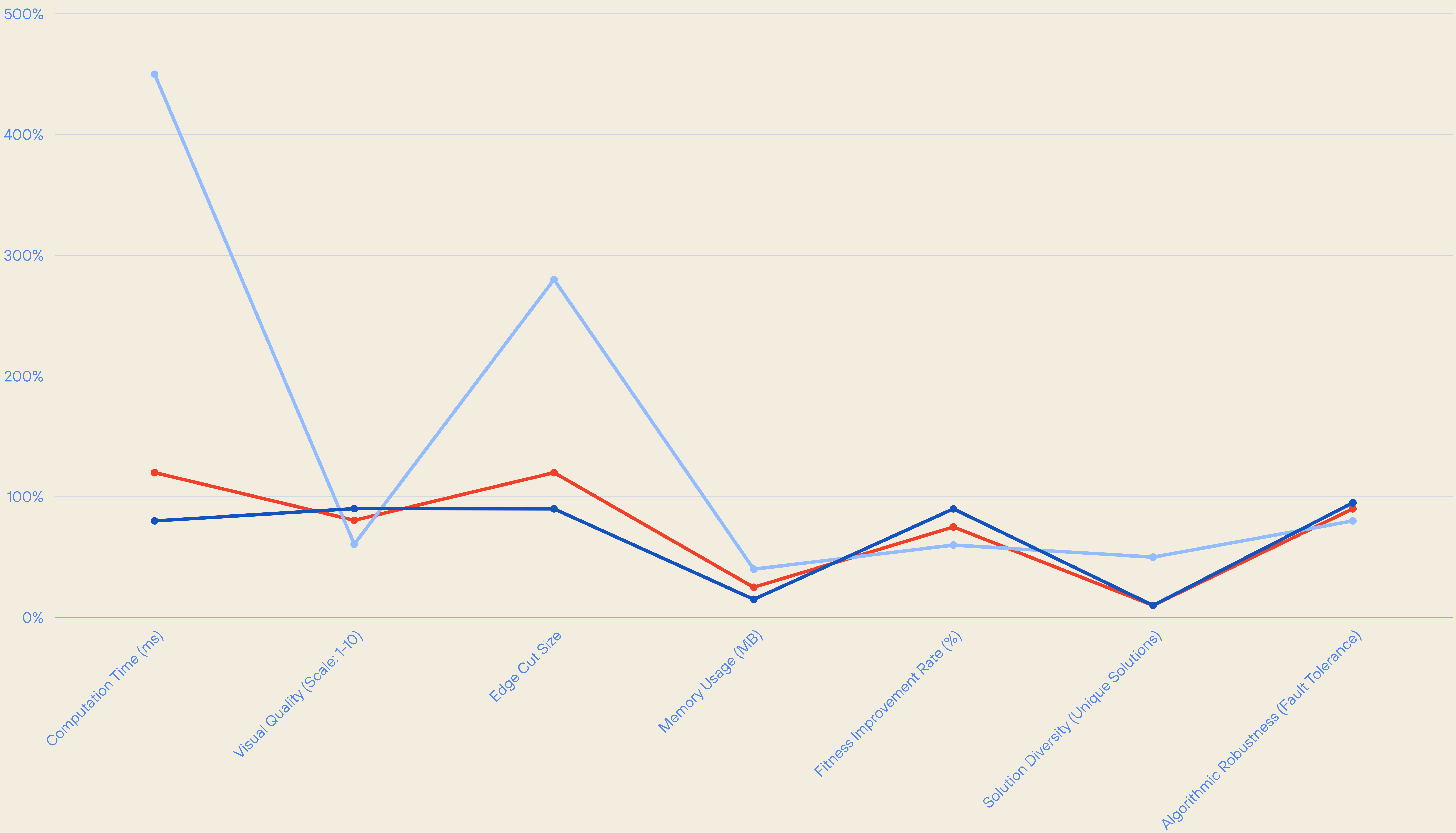
Visual Quality is a subjective measure on a scale from 1 to 10, where higher values indicate better visual quality.



Convergence Speed

Convergence Speed indicates the number of iterations each algorithm took to converge to a solution.




500%
400%
300%
200%
100%
0%
Computation Time (ms)
Visual Quality (Scale: 1-10)
Edge Cut Size
Memory Usage (MB)
Fitness Improvement Rate (%)
Solution Diversity (Unique Solutions)
Algorithmic Robustness (Fault Tolerance)

Summary

- **Spectral Partitioning:**
  - Strengths: High solution quality, effective for large sparse graphs.
  - Considerations: Requires expertise, potentially slow for very large graphs.
- **Genetic Algorithm:**
  - Strengths: Adaptable, suitable for various scenarios.
  - Considerations: Parameter tuning, stochastic nature.
- **Local Search Heuristic:**
  - Strengths: Simplicity, moderate speed for moderately sized graphs.
  - Considerations: Locally optimal, sensitive to initial conditions.