Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

BACHELORTHESIS

Application of Fixed Set Search to the Maximal Partitioning Problem of Graphs with Supply and Demand (MPGSD)

presented by

Manuel Roman Reinhart

Faculty:	MIN-Faculty
	Department of Informatics
Course of Studies:	Information Systems
Matriculation Number:	
Supervisor:	Prof. Dr. Stefan Voß
Primary Referee:	Prof. Dr. Stefan Voß
	Institute of Information Systems University of Hamburg
Secondary Referee:	Dr. Xiaoning Shi Institute of Transport Research, German Aerospace Center

List of Contents

List of Figures

List of Tables

1 Introduction

Graph partitioning is a fundamental problem in computer science, with applications spanning areas such as data mining, network analysis, and the optimization of computational resources. At its core, graph partitioning involves dividing a graph's vertices into multiple groups while minimizing the number of edges between different groups. Such partitioning is crucial for enabling efficient computations on large-scale graphs, particularly in applications that require complexity reduction, such as scientific simulations and network analyses (cf. Buluç et al. (2016)). Graph partitioning is NP-complete, making exact solutions and approximations difficult to achieve (cf. Fan et al. (2020)).

This problem becomes particularly challenging when the graph's vertices represent entities with specific supply or demand attributes, which need to be covered. The Maximal Partitioning Problem of Graphs with Supply and Demand (MPGSD) extends traditional graph partitioning by introducing specific supply and demand attributes of vertices. Unlike standard partitioning, the MPGSD requires that each partition, containing one supply vertex, covers at least the demand of its demand vertices. Since not all nodes might be included, it differs from common partitioning problems; this adaptation is used in applications such as electrical grid management, where each sub-network or partition must be self-sufficient, reflecting a miniaturized yet complete grid system capable of sustaining itself without external support (cf. Jovanovic et al. (2015)).

Despite the extensive studies on graph partitioning, the MPGSD remains computationally challenging due to its NP-hard nature (cf. Ito et al. (2008)), prompting the need for heuristic and metaheuristic approaches. Among these, the Fixed Set Search (FSS) algorithm emerges as a promising approach. Originally designed for combinatorial optimization problems, FSS strategically explores the solution space, generated through a greedy algorithm, by finding reoccurring combinations in the best generated solutions, known as the "fixed set". These fixed sets are then optimized iteratively to approach a near-optimal solution (cf. Jovanovic et al. (2023)).

In recent papers, the FSS has successfully been applied to solve the traveling salesman problem (Jovanovic et al. (2019)) or the clique partitioning problem (Jovanovic et al. (2023)). These applications highlight the ability of the FSS to iteratively refine solutions by identifying and exploiting recurring patterns, making it a powerful tool for complex optimization tasks.

Parallel to this, recent research on solving the MPGSD has explored a variety of methods. Mixed Integer Programming (MIP) has been employed for sparse graphs (Jovanovic and Voß (2016)), where its exact optimization capabilities excel under lower complexity constraints (cf. Bertsimas and Tsitsiklis (1997)). Heuristic local search and correction techniques (Jovanovic et al. (2015)) provide flexible and adaptable approaches that incrementally improve solutions, while metaheuristic methods like Ant Colony Optimization (ACO) (Jovanovic et al. (2016)) utilize pheromone-based probabilistic mechanisms to explore new paths and exploit known solutions, effectively balancing solution quality and computational efficiency (cf. Dorigo et al. (2006)). These methods highlight the ongoing effort to balance computational efficiency and accuracy in solving the MPGSD, each offering unique strengths tailored to specific problem settings.

In this thesis we aim to adapt the FSS to the MPGSD problem, seeking to develop an efficient solution for dividing supply and demand within graph partitions. By leveraging the ability of the FSS to identify recurring patterns across various graph sizes, from small to large, and from low to high connectivity, this approach attempts to balance accuracy and computational efficiency while addressing the challenges posed by supply-demand constraints in graph partitioning.

The thesis is structured as follows. Section 2 provides a detailed and formal definition of the MPGSD. In Section 3, we introduce various solution approaches for solving the MPGSD, including their algorithmic design, such as a greedy approach and the FSS. Section 4 then presents the experimental setup, covering the generation of test instances, evaluation of results and a comparison with an existing approach from the literature. Finally, Section 5 concludes the thesis with a summary of the key findings and suggestions for future research.

2 Problem formulation

In this section, we are going to introduce the Maximal Partitioning of Supply and Demand Graphs (MPGSD) problem. Let us define $G = (V, E)$ to be an undirected graph with a vertex set V and an edge set E. The set V furthermore is partitioned into two disjunct subsets V_s and V_d. Let $|V| = n$, $|V_s| = n_s$ and $|V_d| = n_d$, then $n = n_s + n_d$. Each vertex $u \in V_s$ will be called a supply vertex and is assigned a positive integer $sup(u)$, called a *supply of v*, while each vertex of the second subset $v \in V_d$ will be called a demand vertex and is assigned a positive integer $dem(v)$, called a *demand of v* (cf. Ito et al. (2009)).

As mentioned in Jovanovic et al. (2018), the goal of the MPGSD is to find a set of disjunct subgraphs $\Pi = \{S_1, S_2, S_3, ..., S_n\}$ of the graph G. Each subgraph S_i should contain exactly one distinct supply node $u \in V_s$ and potentially multiple demand nodes. The objective is to maximize the total covered demand over all subgraphs S:

$$max \sum_{S \in \Pi} \sum_{v \in S \cap V_d} dem(v)$$

regarding the following constraints $\forall S_i \in \Pi$:

Each must have a supply greater or equal to its total demand.

$$\sum_{v \in S_i \cap V_s} sup(v) \geq \sum_{v \in S_i \cap V_d} dem(v) \tag{1}$$

Each demand and supply vertex can only be an element of one subgraph; therefore, each demand vertex can only be covered by one supply vertex.

$$S_i \cap S_j = \emptyset, \quad \forall i, j \in \{1, 2, ..., n\} \quad and \quad i \neq j \tag{2}$$

All the subgraphs in Π must be connected.

$$S_i \quad is \quad connected \tag{3}$$

Each subgraph contains only a single supply vertex.

$$\sum_{u \in S \cap V_s} = 1 \tag{4}$$

Note that not all demand vertices are required to be covered by Π.

Figure 2.1 presents an example of the maximal partitioning problem in supply and demand

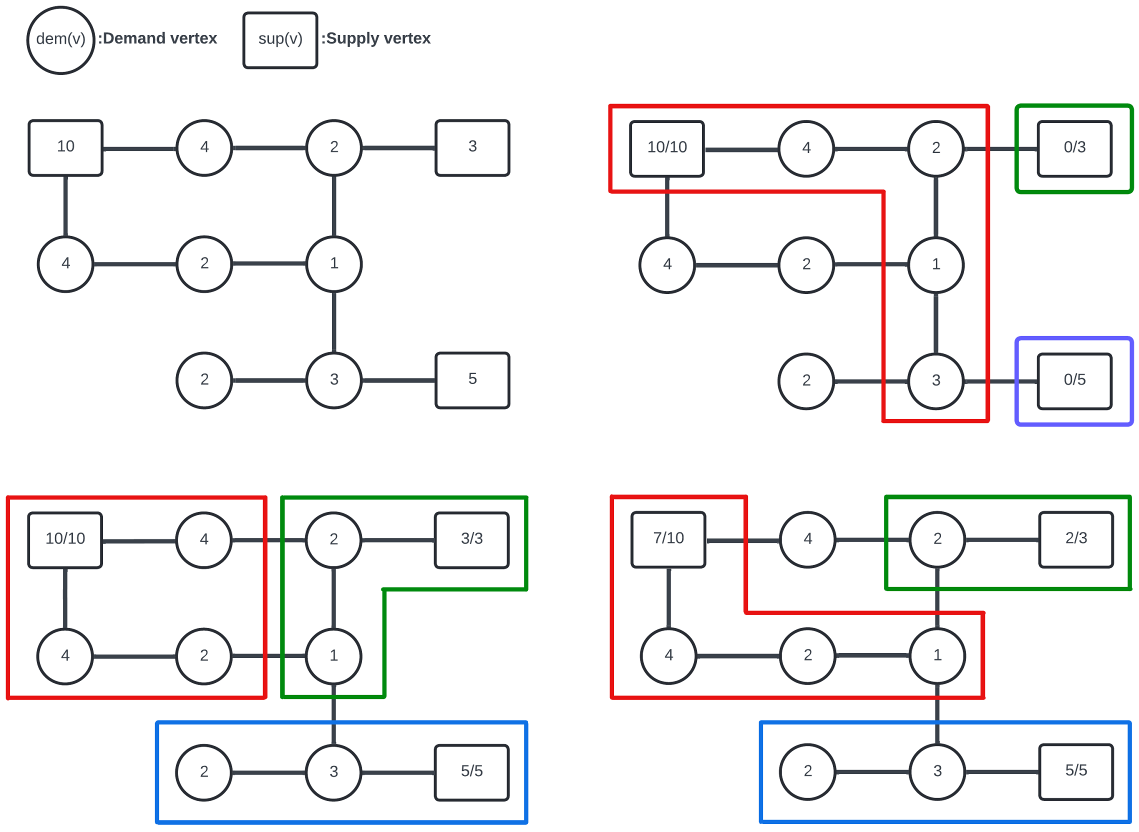

Figure 2.1.: Example of Maximal Partitioning of Supply and Demand Graphs

graphs. The graph in the top left corner shows the initial setup with specific supply and demand vertices. The following image displays various possible partitions, following the mentioned restrictions. Each partition is represented by a unique color to highlight the potential distribution of supply and demand. Even this small example illustrates how many possible partitions can be found even in a simple graph. To determine an optimal solution, it would be necessary to explore every conceivable combination of supply and demand vertices, which confirms the exponential complexity of this problem. Therefore, we aim to find a near-optimal solution using various heuristic approaches.

3 Solution approach

This section outlines the methodologies employed to tackle the maximal partitioning problem of supply and demand graphs. Given the complexity and the combinatorial nature of the problem, we explore various strategies that aim to efficiently find near-optimal solutions. The methods discussed include a Greedy algorithm, which offers a straightforward approach to partitioning by iteratively making local optimal choices. Additionally, we explore a matheuristic approach that combines heuristic methods to improve both solution quality and computational efficiency. This approach is detailed through its key components: the Fixed Set Search and an Enhanced Greedy Algorithm, which incorporates the initialization using the fixed set for better performance and stability to tackle the problems complexity.

3.1 Greedy algorithm

To find an initial solution for the problem and subsequently create multiple solution instances to identify recurring fixed sets, we start by implementing a greedy algorithm. The key principle of greedy algorithms is that at each step, decisions are made by selecting the locally optimal choice (e.g., adding the best adjacent vertex to a subgraph) without reconsidering previous decisions. Once a decision is made, it is final, and the algorithm does not backtrack or reverse these choices (cf. Cormen et al. (2022)).

We will start our greedy algorithm with a given MPGSD graph G. The solution will contain $n = |V_s|$ subgraphs, one for each supply vertex. After initialization each subgraph will only contain its corresponding supply vertex. Afterwards the subgraph S_i with the most remaining supply, will be selected and extended by a vertex $v \in V_d$, which is adjacent, not yet part of any other subgraph and whose demand can be fulfilled by the supply vertex. The choice of vertex can be determined depending on the desired trait, similar to those mentioned by Jovanovic et al. (2015):

- **Trait 1:** Most available demand that is not covered
- **Trait 2:** Most adjacent vertices
- **Trait 3:** A combination of adjacent vertices and remaining demand
- **Trait 4:** A random trait from the above

If selecting a vertex by the first trait, an iteration is performed over each vertex $u \in S_i$ in the current subgraph. For each vertex u, we look at its adjacent vertices $v \in V_d$. The max $dem(v) \quad \forall v \in V_d$ will be selected if it exceeds the demand of previously found vertices and if its demand can be fulfilled. After going over every vertex u and its adjacent vertices, the vertex with the maximum demand will be returned. This heuristic is effective, if we want to cover high demand relatively quickly. However, subgraphs might block other subgraphs from further expansion, locking them in place. To tackle this challenge we take a look at the second trait.

If we select a vertex using the second trait, we follow a similar approach, but instead of selecting the vertex with the most demand, we select the vertex with the largest number of adjacent vertices, whose demands are not yet being covered. This way, we assure that the subgraph has enough "room" to expand to further vertices and does not get stuck (as illustrated in the top right partition of Figure 2.1).

To embrace both traits, we can use a combination of both. Let $x = \frac{\text{dem}(v)}{|adj(v)|+1}$ with $|adj(v)|$ being the number of adjacent vertices of v, whose demands are not yet being covered. We define our ratio $r = \frac{1}{x}$. We then select the vertex with the maximum r among all vertices found and add it to the subgraph.

For example, let us assume we have a vertex k with $dem(k) = 10$ and $|adj(k) = 1|$ and a vertex p with $dem(p) = 10$ as well, but $|adj(p)| = 3$. For k our $r = 0.2$ and for p our $r = 0.4$. In this scenario, vertex p would be selected as our maximum and added to the subgraph.

Finally, instead of generating just one solution each time we solve our MPGSD graph G, we can obtain multiple solutions by using a random trait from the three available traits for selecting each new vertex to add to the subgraph. This approach allows us to gather a diverse set of solutions, which can later be analyzed to identify recurring fixed sets.

If no fitting adjacent vertex can be found in any of the subgraphs, due to a lack of remaining supply or because no adjacent vertices are available, the algorithm terminates and returns Π with a set of subgraphs $S_i \in \Pi$ for G. At this stage, we can consider our graph G solved. However, it is important to note that we cannot guarantee that our solution is optimal or fully satisfactory, i.e., all supply is utilized or all demand is covered.

3.1.1 Pseudocode for greedy algorithm

If we take a look at the pseudocode of algorithm 1: *greedySolve*, we see that the algorithm takes an MPGSD graph and one of our previously discussed traits. The algorithm will return a number of subgraphs that represent our solution, similar to the colored sections in Figure 2.1.

We begin by resetting the vertices of our graph. This step ensures that our graph can be solved multiple times without needing to build a new graph each time. This becomes important when discussing our Fixed Set Search, which requires solving the problem multiple times. After resetting each vertex, we create a new subgraph for each supply vertex in our MPGSD graph.

Next, we initialize the covered demand of the entire problem to 0 and set the total remaining supply to the combined supply of the supply vertices. Then, we enter a loop in line 4 that always selects the subgraph with the highest remaining supply that is not yet complete. If a subgraph is found that is not yet finished, we select the best fitting vertex to add to our subgraph using our selected trait. A vertex can only be selected if its demand can be fulfilled by the supply vertex of the subgraph, it is adjacent to any vertex of the subgraph, and it is not already covered by another subgraph. If these conditions are not met, the subgraph is set to complete. Otherwise, the selected vertex is added to the subgraph. We then update the vertex to be set as covered and adjust the remaining supply of the subgraph, as well as the overall used supply and covered demand.

The loop ends when no subgraph is available for selection. The algorithm then returns its solved graph containing our subgraphs and terminates.

Algorithm 1 greedySolve: Pseudocode for solving the MPGSD problem using a greedy approach.

```
Input: $g$: MPGSD Graph, $trait$: trait for vertex selection
Output: A solved graph containing all the subgraphs, the $graphOfSubGraphs$
 1: resetGraphVertices($g$)
 2: $graphOfSubGraphs \leftarrow$ new SolvedGraph($g$)
 3: $graphOfSubGraphs.updateSupAndDem(g.getSupAndDem())$
 4: while true do
 5:     $selectedSubGraph \leftarrow graphOfSubGraphs.getSubgraphWithHigestSupply()$
 6:     if $selectedSubGraph$ is null then
 7:         break
 8:     end if
 9:     $selectedAdjDemV \leftarrow selectedSubGraph.getVertexToAdd$(trait)
10:     if $selectedAdjDemV$ is null then
11:         $selectedSubGraph.setComplete()$
12:     else
13:         $selectedAdjDemV.updateVertex()$
14:         $selectedSubGraph.addVertex(selectedAdjDemV)$
15:         $graphOfSubGraphs.updateSupAndDem(selectedAdjDemV.getDemand())$
16:     end if
17: end while
18: return graphOfSubGraphs
```

3.2 A matheuristic based on the fixed set search

To address the problem, that we cannot guarantee that our solution is optimal, and achieve more satisfactory results, we implement a fixed set search algorithm. The fundamental concept of this approach is to solve the graph multiple times, identify instances of recurring vertex combinations, and initialize our greedy algorithm again using the discovered combinations.

This method allows us to leverage previously identified, high-potential, patterns and avoid starting the entire process from scratch each time. By doing so, we aim to enhance both the efficiency and the quality of the solutions produced. This allows the algorithm to focus on optimizing the remaining parts of the graph rather than starting from scratch each time (cf. Jovanovic et al. (2023)).

3.2.1 Fixed set search

As mentioned before, a fixed set is a subset of vertices that frequently appear in high-quality solutions. At the start of Algorithm 2: *getFixedSets*, we initialise the fixed sets as an empty list, where we will later on store one subgraph for each corresponding supply vertex. We also initialize an empty array of size m, where we store the m_{best} solutions, which will later be analysed to find recurring patterns.

After initialization, we greedily solve our MPGSD problem t times, with $t \geq m_{best}$, using a random trait (trait 4), as described in Section 3.1, to create a variety of solutions which differ from one another. If a solution passes a certain threshold:

$$\frac{Supply_{covered}}{Supply_{total}} > threshold$$

which can be defined beforehand, we consider it a good enough solution to be added to our array of m_{best} solutions. If the array of m_{best} solutions is already full, a new solution that outperforms any of the existing ones in terms of demand coverage will replace the current worst-performing solution. If none of the t solutions surpasses the threshold, we simply add a solution where each subgraph contains only its supply vertex.

Once we have our array of m_{best} solutions, we aggregate all subgraphs for each supply vertex together, as shown in line 16 of Algorithm 2. For example, all the subgraphs corresponding to supply vertex one from every solution in our array are grouped together. This results in m subgraphs per supply vertex. Before we proceed with finding the common subgraphs, we need to reset the graph vertices because they will be rebuilt during the creation of the fixed sets.

The algorithm for finding the fixed sets, as represented in line 19 of Algorithm 2, is detailed in Algorithm 3: *findFixedSet* and will be explained in the next paragraph. It involves analyzing these grouped subgraphs to identify recurring patterns and create a common subgraph for each supply vertex. After finding a common subgraph for each supply vertex, using Algorithm 3, it is added to the list of fixed sets. Once all groups of subgraphs have been processed, the algorithm returns the complete list of fixed sets.

Algorithm 2 getFixedSets: Pseudocode for getting fixed sets

```
Input: g: MPGSD Graph, t: number of greedy solutions, m: number of solutions to
    consider for FSS, threshold: percentage of supply that should be covered
Output: A list of subgraphs, one for each supply vertex, the fixedSets
1: fixedSets ← empty list
2: arrayOfBestGreedySolutions ← new array of size m
3: for i ← 0 to t do
4:     solution ← greedySolve(g, 4)
5:     covSup ← solution.getCovSup()
6:     totalSup ← solution.getTotalSup()
7:     supPercentCovered ← $\frac{covSup}{totalSup}$
8:     if supPercentCovered > threshold then
9:         findPlaceInArray(arrayOfBestGreedySolutions, solution)
10:    end if
11: end for
12: if arrayOfBestGreedySolutions is empty then
13:     replacementSolution ← new SolvedGraph(g)
14:     arrayOfBestGreedySolutions[0] ← replacementSolution
15: end if
16: listForEachSupply ← sortBySupplyV(arrayOfBestGreedySolutions)
17: resetGraphVertices(g)
18: for x ← 0 to listForEachSupply.size() − 1 do
19:     fixedSet ← findFixedSet(listForEachSupply.get(x), g)
20:     fixedSets.add(fixedSet)
21: end for
22: return fixedSets
```

In Algorithm 3, for finding each fixed set out of our list of aggregated supply vertices, we begin by examining all the edges in each subgraph and count their occurrences, storing their sum in a frequency map. If an edge appears in more than 50% of the subgraphs, it is considered common and included in a new subgraph called the *commonSet*. We then create our final subgraph, the *fixedSet*, by applying the *extractConnectedComponent* method to the *commonSet*. This method checks the connectivity of the *commonSet* using a Depth-First Search (DFS) and returns only the portion of the *commonSet* that is connected within the same cluster as the supply vertex. Finally our algorithm then terminates and returns this *fixedSet*. If no common edges are found initially, we return a subgraph containing only the supply vertex. By repeating this process for each of the aggregated supply vertices, we obtain one fixed set per supply vertex, collectively forming our fixed sets.

Algorithm 3 findFixedSet: Pseudocode for finding a fixed set out of multiple subgraphs

```
Input: $subgraphsForOneSupply$: list of subgraphs, $g$: MPGSD Graph
Output: A common subgraph for the corresponding supply vertex, the $fixedSet$
 1: $edgeFrequency \leftarrow$ empty map
 2: for $i \leftarrow 0$ to $subgraphsForOneSupply.size() - 1$ do
 3:     countAllEdges($subgraphsForOneSupply.get(i)$, $edgeFrequency$)
 4: end for
 5: $threshold \leftarrow subgraphsForOneSupply.size()/2$
 6: $commonEdges \leftarrow$ empty set
 7: for each entry in $edgeFrequency$ do
 8:     if entry.value $>$ threshold then
 9:         $commonEdges.add(entry.key)$
10:     end if
11: end for
12: if not $commonEdges.isEmpty()$ then
13:     $commonSet \leftarrow$ createFixedSet($commonEdges$)
14:     $fixedSet \leftarrow commonSet.extractConnectedComponent()$
15:     return $fixedSet$
16: end if
17: return new SubGraph($getSubgraphsSupplyVertex()$)
```

Let us illustrate the basic concept of finding a fixed set with a small example. As seen in Figure 3.1, we have two different solutions for the same MPGSD problem. Let us assume they would surpass our initial threshold and are considered good solutions. Let $m = 2$, and these two solutions are in our array of m_{best} solutions.

In Algorithm 3, we count all the edges for the first supply vertex, in this case, the one surrounded red. We would count the edges from sup(10) to dem(4) and from dem(4) to dem(2) twice. The edge from sup(10) to the upper vertex dem(4) and the edge from dem(2) to dem(1) are only counted once.

Because only the edges from sup(10) to dem(4) and from dem(4) to dem(2) occur in over 50% of the solutions, they are considered common. Therefore, our corresponding fixed set for the supply vertex surrounded in red would include the edges from sup(10) down to dem(4) and from dem(4) to dem(2).

After repeating the process for every group of supply vertices, such as the green and blue surrounded supply vertices in our example, we would obtain the shown fixed sets.

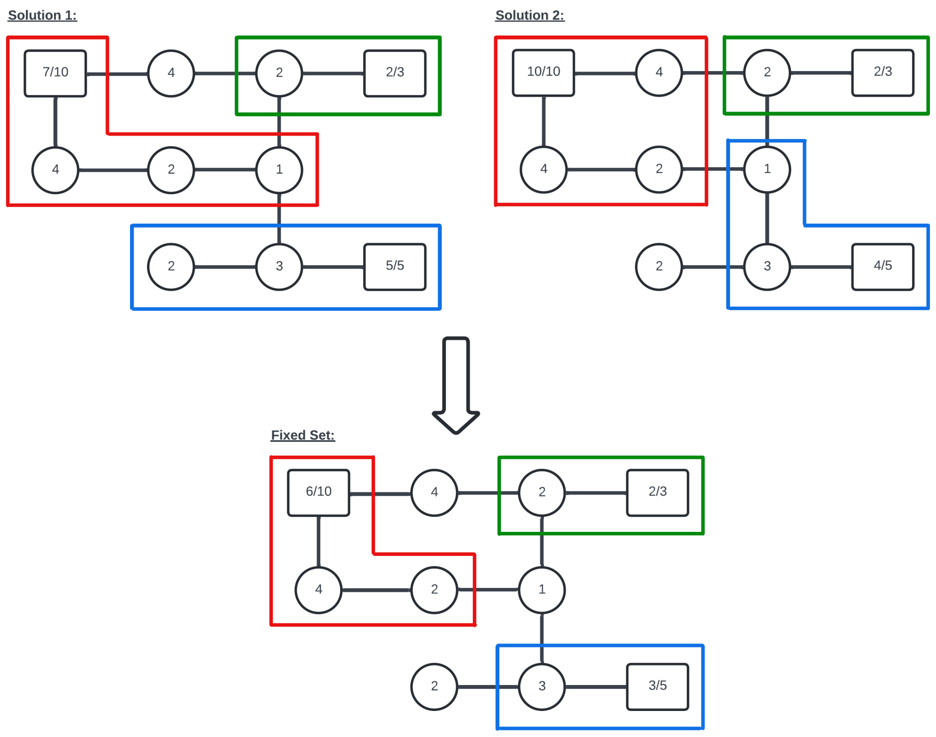

Figure 3.1.: Example of a Fixed Set

3.2.2 Enhanced greedy algorithm with fixed set initialization

To solve our MPGSD problem using the identified fixed sets, we use Algorithm 4. We start by initializing *currentBestCoverage* to zero and set *bestGraph* null. The algorithm then iterates a specified number of times. In each iteration, we reset the graph vertices from previous solutions, rebuild the fixed sets and greedily solve our graph as described in Algorithm 1, but with a modified initialization. Instead of starting each subgraph from scratch with only its supply vertex, we initialize each subgraph with the vertices from the fixed set, as shown in line 6 of Algorithm 4. If the demand coverage in the current solution is higher than our current best coverage, we update *bestGraph* and *currentBestCoverage* to the current solution's values.

Trait 4 includes randomness by alternating between the different greedy strategies (Traits 1, 2, and 3). This randomized greedy algorithm with pre-selected elements aims to explore a wider range of potential solutions by varying the method of vertex selection. The purpose of performing multiple iterations with the previously identified fixed set is to diversify the search process, allowing the algorithm to explore various solution paths and ultimately select one with the most demand coverage out of all solutions.

By using the "Fixed Set Search" approach to solve our problem, we anticipate the following benefits:

Algorithm 4 getBestFSSolution: Pseudocode for solving the MPGSD problem using a greedy approach with fixed sets.

Input: *iterations*: number of iterations, *g*: MPGSD Graph, *trait*: trait for vertex selection, *fixedSets*: list of fixed sets
Output: A solved graph to the MPGSD problem using Fixed Sets, the *bestGraph*

```
1: $currentBestCoverage \leftarrow 0$
2: $bestGraph \leftarrow$ null
3: for $i \leftarrow 1$ to $iterations$ do
4:     resetGraphVertices($g$)
5:     rebuildFixedSetVertices($fixedSets$)
6:     $solved \leftarrow$ greedySolve($g$, $trait$, $fixedSets$)
7:     $currentDemCov \leftarrow solved.getTotalCoveredDemand()$
8:     if $currentBestCoverage < currentDemCov$ then
9:         $currentBestCoverage \leftarrow currentDemCov$
10:        $bestGraph \leftarrow solved$
11:    end if
12: end for
13: rebuildFixedSetVertices($bestGraph$)
14: return $bestGraph$
```

1. **Efficiency:** Starting the greedy algorithm with pre-selected fixed sets reduces the need for to start from scratch with only a single supply vertex each time, leading to faster high-quality solutions. By leveraging previously identified successful vertex combinations, the algorithm can bypass many of the early, less effective iterations.

2. **Improved solution quality:** Due to the fixed sets, the algorithm starts in more promising regions of the solution space, improving the chances of finding near-optimal solutions for the MPGSD problem. Reusing successful vertex combinations from previous runs can therefore enhance the overall quality of the method.

However, there are potential drawbacks to this approach:

1. **Dependency on initial solutions:** The effectiveness of the fixed set search depends heavily on the quality of the initial solutions. If the initial solutions are not are not diverse or of high enough quality, the fixed sets identified will also be insufficient, which can lead to a suboptimal final solution.

2. **Limited exploration:** While the fixed sets may lead to starting in promising regions, as mentioned before, this could also limit the further exploration of possible solutions and potentially result in missing out on solutions which do not fit to our initial partition.

In the next section, we are going to analyse how effectively the Fixed Set Search approach performs across different types of graphs to determine whether these benefits hold true or whether the potential drawbacks influence the success of the algorithm.

4 Results

In this section, we present the results of our computational experiments to evaluate the effectiveness of the proposed solution approach for the MPGSD. We begin by describing the generation of test instances, followed by the experimental settings, including the hardware and software environment used for the experiments. Finally, we present and analyze the results of our computational experiments, focusing on the influence of different variables, such as the number of iterations, the amount of m_{best} solutions or the different solving traits, in terms of solution performance and computational time. Additionally, we compare our method with a correction approach proposed by Jovanovic et al. (2015) to assess the effectiveness of our approach against existing methods.

4.1 Test instance generation

To create the test instances for the MPGSD problem, we convert a JSON file into our graph. The JSON file follows a specific structure:

Listing 1 JSON for MPGSD graph

```json
{
    "supplyVertices": [
      {"id": 1, "value": 10}
    ],
    "demandVertices": [
      {"id": 2, "value": 2},
      {"id": 3, "value": 4},
      {"id": 4, "value": 3},
      {"id": 5, "value": 3},
    ],
    "adjacencies": [
      {"source": 1, "targets": [2]},
      {"source": 2, "targets": [3, 4, 5]},
    ]
  }
```

Each of the supply and demand vertices is initialized with its corresponding value. Supply vertices are initialized with their supply values, and demand vertices with their demand values. The adjacencies are then constructed based on the JSON structure.

For example, an entry such as "source": 2, "targets": [3, 4, 5] means that edges are created between vertex 2 and vertices 3, 4, and 5. Consequently, vertices 3, 4, and 5 are added to the adjacency list of vertex 2, and vertex 2 is added to the adjacency lists of vertices 3, 4, and 5. By processing every entry in the adjacencies list, we construct the entire MPGSD graph.

The JSON file is generated using a Python script. First, we define the number of supply vertices, num_{sup}, and demand vertices, num_{dem}, that we want our graph to consist of. Each demand vertex is assigned a value within the interval $[10, 40]$. The interval for the supply values $[\min_{sup}, \max_{sup}]$ is calculated by considering the average demand, as well as the number of demand and supply vertices. The upper limit of the supply value interval, $\max_{sup}$, is calculated using the formula:

$$\max_{sup} = avg_{dem} \times \frac{num_{dem}}{num_{sup}}$$

The result defines the upper boundary of the supply values. To determine the lower boundary, min_{sup}, we divide max_{sup} by 10:

$$\min_{sup} = \frac{\max_{sup}}{10}$$

After having set the values of the vertices, we need to create the adjacencies. To evaluate our results effectively, it is crucial that we know what the best possible solution would look like beforehand. We begin by selecting a supply vertex and adding a random demand vertex to its adjacencies. If the supply is not fully covered by this vertex, we continue by adding a demand vertex to the adjacnecies of the previous one, that has a demand less than the remaining supply. Continuing this process creates a chain of demand vertices, creating a snake-like pattern. If the remaining supply is lower than the minimum demand value min_{dem}, we add a final demand vertex whose demand is set equal to the remaining supply, thus making an exception to the preset interval boundaries, but it allows us to guarantee that there is an optimal solution that utilizes the entire supply.

After applying this process to each supply vertex, the optimal solution is equal the total available supply. This allows us to compare the total available supply of the MPGSD graph to the supply coverage of our FSS results and calculate the relative errors of the generated solutions.

To complete the adjacencies of the entire graph, we differentiate between low and high connectivity graphs. In a highly connected graph, each vertex has 10 random vertices added to its targets, whereas in a low connectivity graph, each vertex will have only one random vertex added to its targets. To ensure connectivity, we perform a Depth-First Search (DFS) to check the graph's connectivity. If any vertex is not connected to the rest of the graph, a random connection is added to its targets. The DFS is repeated until all vertices are part of a single connected graph. This guarantees that there is a path between any two vertices, ensuring the MPGSD graph is fully connected.

4.2 Experimental settings

This section outlines the computational environment and settings employed for testing and evaluating the performance of our algorithm.

The algorithms were developed in Java using the Eclipse IDE 2024-06. All experiments were executed on a machine featuring an AMD Ryzen 5800X 8-Core Processor, paired with 32 GB of DDR4-3000 MHz RAM, running Microsoft Windows 10 64-bit. Although the system is equipped with 32 GB of RAM, Eclipse was configured to utilize a maximum of 2 GB during testing to maintain consistent performance.

To provide a thorough evaluation of the proposed algorithm, we tested it on a range of graph sizes. The selection of graph sizes was based on the methodology described in Jovanovic et al. (2015), while the implementation of the graphs was done according to the procedures detailed in the previous section 4.1.

4.3 Computational experiments

As test graphs, we use both highly connected and low connected MPGSD graphs, as mentioned in Section 4.1. To assess our results as accurately as possible, we generated 500 solutions for each test case to determine the average from these solutions. We evaluate all results based on the following metrics:

- **Average Relative Error (Avg) in percent**:

$$\text{avg}_{\text{relativeError}} = \left(1 - \frac{\text{avg}_{\text{coveredDem}}}{\text{optimal}}\right) \times 100$$

- **Standard Deviation (SD) in percent**:

$$\text{SD} = \sqrt{\frac{\sum(\text{relativeError} - \text{avg}_{\text{relativeError}})^2}{\text{solutions}}} \times 100$$

- **Maximum Relative Error (Max) in percent**.

Due to the method of the FSS, we have a lot of different variables, we can change, be it the amount greedy iterations we perform to find our m_{best} solutions, the size of our m_{best} solutions to create our fixed set, or the greedy solving trait, we use to solve the problem using our fixed set. For all tests, we kept the threshold fixed at 0.7, meaning only solutions with a supply coverage greater than 70% were considered for our array of m_{best} solutions.

The first table 4.1 shows the results if we were to solve the MPGSD graph using only our normal greedy traits, without performing a FSS. For each trait, the results are listed for both, low and high connected, graphs.

From our results, we can conclude that trait 1 for highly connected graphs outperforms the other two traits for every graph. For low connected graphs, trait 1 performs worse than traits 2 and 3 for some graphs, but still outperforms both traits for most low connected graphs. Therefore, after generating our fixed set, we use trait 1 in the following tables, to determine our final results, in order to better compare the influence of changing each variable, such as the amount iterations or the size of m_{best}.

When examining the influence of the number of m_{best} solutions, it is evident that increasing the number of m_{best} solutions generally leads to a lower Maximum Relative Error and Standard Deviation in almost all cases, for both low connected and highly connected graphs. This trend indicates that a larger pool of solutions contributes to greater stability and consistency in the results. However, the impact on the Average Relative Error is more nuanced. In highly connected graphs, the best results for Average Relative Error are often achieved with $m_{best} = 10$. For low connected graphs, the outcomes are more varied; while increasing m_{best} often outperforms some solutions with a smaller amount of m_{best} solutions by a few percentage points, there can be large jumps in error rates, as seen in Table 4.3. When taking a look at the 2x40 graph, the Average Relative Error increases from 7.36% with $m_{best} = 10$ to 17.86% with $m_{best} = 100$.

In assessing the impact of the number of iterations, the data clearly shows that more iterations consistently decrease the Maximum Relative Error, Standard Deviation and in nearly all cases, reduce the Average Relative Error as well. This applies for both low connected and highly connected graphs, reinforcing the idea that more iterations lead to a greater exploration of the solution space, resulting in better overall results.

When comparing the Average Relative Error of the FSS approach to the results from Table 4.1, where the graph was solved using only a greedy approach with a specific trait, we can observe that the FSS approach almost always yields better results in low connected graphs, with improvements in at least 75% of results. This noticeable improvement highlights the effectiveness of the FSS in scenarios where the connectivity of the graph is low and the problem complexity is higher.

Table 4.1.: Comparison of the Average Relative Error for solving the MPGSD problem with the basic greedy traits mentioned in Section 3.1 using highly and low connected graphs

Sup x dem	Error trait 1 (%)		Error trait 2 (%)		Error trait 3 (%)	
	High Conn.	Low Conn.	High Conn.	Low Conn.	High Conn.	Low Conn.
2x6	0.0	0.0	31.82	58.49	4.55	56.60
2x10	0.0	0.0	4.35	14.44	13.04	17.78
2x20	0.94	33.76	13.21	24.12	5.35	14.15
2x40	0.0	25.92	0.89	31.70	3.55	29.25
5x15	0.0	26.63	10.28	38.46	10.28	26.63
5x25	2.83	0.84	14.16	36.59	13.88	19.27
5x50	0.99	18.47	7.12	24.90	7.45	5.09
5x100	0.0	20.62	3.52	22.06	4.58	19.96
10x30	2.78	2.30	23.54	35.25	14.68	11.06
10x50	10.30	7.30	13.54	21.90	17.78	13.77
10x100	1.40	25.27	7.44	28.29	8.29	13.89
10x200	0.20	8.94	5.29	30.57	3.35	27.12
25x75	3.26	4.74	23.21	14.85	14.82	6.62
25x125	1.06	6.08	19.35	17.87	13.40	9.51
25x250	1.83	5.34	14.48	10.40	7.58	9.01
25x500	0.66	2.42	4.10	16.74	6.07	10.36
50x150	2.15	2.96	22.67	21.93	15.02	4.06
50x250	2.69	5.54	16.77	20.29	14.61	5.25
50x500	2.35	5.26	9.70	12.24	7.93	6.78
50x1000	0.61	8.52	3.23	19.05	2.77	9.62
100x300	5.55	1.27	27.63	18.71	14.96	3.27
100x500	4.06	5.06	13.31	15.37	14.22	4.05
100x1000	1.01	4.07	7.34	9.10	8.41	5.73
100x2000	0.50	4.66	3.47	11.37	3.76	5.97
200x600	3.84	1.12	24.44	17.52	13.30	1.18
200x1000	4.44	3.09	14.63	11.11	14.65	3.11
200x2000	2.01	4.91	7.18	7.24	7.78	3.25
200x4000	0.34	3.95	2.91	9.23	3.81	4.14
400x1200	4.74	1.04	24.46	15.19	12.71	0.88
400x2000	4.51	3.79	16.29	10.49	13.02	2.06
400x4000	1.10	4.20	6.70	6.79	8.11	3.50
400x8000	0.40	3.68	3.26	6.39	3.97	3.58

In contrast, in highly connected graphs where trait 1 performs exceptionally well, the FSS still manages to outperform it in around 50% of the cases, particularly when a lower percentage of m_{best} solutions is used relative to the number of iterations.

Table 4.2.: Comparison of Fixed Set Search performance in highly connected graphs by varying the number of m_{best} solutions considered for Fixed Set construction (selected from 500 greedy iterations)

Sup x dem	$m_{best} = 10$			$m_{best} = 25$			$m_{best} = 50$			$m_{best} = 100$		
	Avg	Max	SD	Avg	Max	SD	Avg	Max	SD	Avg	Max	SD
2x6	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2x10	0.00	0.00	0.00	0.00	0.00	0.00	1.57	1.57	0.61	0.00	0.00	0.00
2x20	0.00	8.49	1.20	0.00	5.03	0.83	1.77	2.44	0.80	1.57	1.57	0.61
2x40	0.00	3.77	0.86	0.67	3.55	0.81	0.93	2.34	0.87	0.22	2.00	0.80
5x15	0.00	0.93	0.37	0.93	2.34	0.62	1.13	4.53	0.78	0.00	2.34	0.53
5x25	1.13	4.82	0.80	2.55	3.97	0.74	1.99	4.14	0.79	4.53	6.80	1.30
5x50	3.15	4.14	0.79	0.66	4.14	0.81	0.20	1.33	0.27	1.99	4.14	0.61
5x100	0.13	1.53	0.29	0.27	1.53	0.27	2.03	4.81	0.76	0.00	1.26	0.24
10x30	2.03	4.81	0.50	2.03	4.81	0.95	1.62	4.04	0.62	2.03	4.81	0.88
10x50	2.02	4.24	0.73	0.81	5.45	0.68	1.01	3.02	0.38	2.02	3.84	0.67
10x100	1.32	2.87	0.48	0.93	3.10	0.49	0.20	0.78	0.13	1.55	2.25	0.32
10x200	0.75	0.95	0.15	0.34	0.91	0.15	2.89	8.48	1.22	0.37	0.88	0.12
25x75	1.86	7.64	0.98	2.33	8.48	1.36	2.34	3.84	0.43	4.75	8.48	0.82
25x125	2.17	3.95	0.56	1.61	3.67	0.44	1.34	6.21	0.89	1.78	3.11	0.43
25x250	4.46	5.32	1.32	1.19	2.81	0.38	0.41	0.70	0.12	0.94	5.76	0.69
25x500	0.41	0.87	0.14	0.59	0.83	0.13	0.41	0.70	0.12	0.35	0.74	0.11
50x150	3.44	7.32	0.99	4.97	7.36	0.88	4.88	7.70	0.69	6.07	7.99	0.58
50x250	2.29	5.47	0.71	3.04	4.88	0.65	4.11	4.77	0.71	3.52	4.72	0.77
50x500	1.20	2.34	0.28	1.04	2.17	0.29	0.92	2.38	0.26	0.97	1.95	0.21
50x1000	0.47	0.88	0.11	0.41	0.84	0.11	0.81	0.84	0.11	0.48	0.78	0.10
100x300	3.61	5.47	0.56	3.90	5.79	0.47	4.37	6.08	0.37	4.56	5.71	0.29
100x500	2.45	3.99	0.38	2.53	3.31	0.29	2.35	3.54	0.26	2.78	3.28	0.23
100x1000	1.06	1.48	0.16	1.22	1.50	0.15	0.97	1.50	0.15	1.12	1.48	0.15
100x2000	0.35	0.70	0.07	0.44	0.66	0.07	0.47	0.64	0.07	0.55	0.66	0.07
200x600	3.48	5.68	0.46	3.83	5.97	0.40	5.19	5.62	0.37	4.79	5.63	0.31
200x1000	3.10	3.96	0.32	2.54	3.87	0.30	2.48	3.67	0.27	3.34	3.84	0.24
200x2000	1.24	1.76	0.16	1.35	1.70	0.13	1.19	1.74	0.13	1.12	1.66	0.12
200x4000	0.45	0.76	0.06	0.45	0.63	0.05	0.51	0.61	0.05	0.40	0.60	0.05
400x1200	4.06	5.05	0.33	4.37	4.96	0.30	4.26	4.88	0.24	4.17	4.76	0.21
400x2000	2.59	4.03	0.28	3.45	4.05	0.24	3.29	3.81	0.24	2.99	3.88	0.22
400x4000	1.18	1.48	0.09	1.03	1.42	0.08	1.12	1.34	0.06	1.10	1.27	0.06
400x8000	0.45	0.55	0.04	0.48	0.58	0.04	0.43	0.55	0.04	0.43	0.55	0.03

When considering the number of iterations required to outperform trait 1 in highly connected graphs, the results indicate that with just 50 iterations, the FSS tends to perform worse. At 100 iterations, the results are mixed, with outcomes being mostly evenly split between performing better or worse than trait 1. However, as the number of iterations increases, the FSS starts to outperform the greedy approach more consistently, eventually leading to better results in more than half of the cases. Meaning that even in graphs with high connectivity, the FSS can offer advantages, when carefully selecting the appropriate number of m_{best} solutions and amount of iterations.

Table 4.3.: Comparison of Fixed Set Search performance in low connected graphs by varying the number of m_{best} solutions considered for Fixed Set construction (selected from 500 greedy iterations)

Sup x dem	$m_{best} = 10$			$m_{best} = 25$			$m_{best} = 50$			$m_{best} = 100$		
	Avg	Max	SD	Avg	Max	SD	Avg	Max	SD	Avg	Max	SD
2x6	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2x10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	5.56	0.25	5.56	5.56	2.44
2x20	0.64	3.22	0.94	1.61	3.22	0.79	3.22	3.22	0.27	2.57	2.57	0.06
2x40	7.36	19.44	1.49	7.36	19.44	5.24	19.44	19.44	0.52	17.86	18.91	0.36
5x15	6.51	6.51	0.00	13.02	13.02	1.55	13.02	13.02	1.94	13.02	13.02	0.00
5x25	2.23	5.59	1.00	3.07	5.59	1.09	1.68	4.47	0.76	0.84	3.63	0.78
5x50	6.69	12.99	1.71	5.49	14.86	1.34	5.49	14.86	3.26	11.78	18.07	3.26
5x100	1.00	1.55	0.23	0.33	2.11	0.34	0.67	2.55	0.68	2.00	3.44	0.36
10x30	7.83	7.83	1.43	2.30	7.83	1.23	2.30	7.83	1.31	2.30	5.53	0.38
10x50	0.69	3.17	0.42	0.96	4.27	0.23	3.44	4.41	0.67	4.27	5.65	1.56
10x100	5.24	12.29	1.71	9.68	10.64	0.69	9.68	10.42	0.24	9.90	15.99	1.13
10x200	3.83	4.91	0.84	4.03	8.36	0.67	3.95	8.44	1.35	7.95	8.44	1.46
25x75	4.74	5.01	0.04	4.74	4.83	0.03	4.74	4.83	0.02	4.74	4.83	0.03
25x125	3.71	4.37	0.25	3.87	4.37	0.18	3.71	4.26	0.12	3.93	4.20	0.08
25x250	3.48	5.09	0.61	3.48	4.77	0.50	4.20	4.74	0.40	3.32	5.79	0.31
25x500	3.75	4.63	0.32	3.79	4.63	0.16	3.88	4.47	0.15	3.83	4.54	0.18
50x150	0.75	1.40	0.17	1.30	1.40	0.14	1.00	1.40	0.11	1.30	1.30	0.12
50x250	1.87	4.15	0.57	2.25	3.48	0.43	1.49	3.07	0.37	1.42	3.07	0.34
50x500	2.98	3.76	0.36	3.10	3.75	0.27	3.29	3.91	0.23	3.17	3.95	0.20
50x1000	5.19	5.76	0.14	5.30	5.74	0.11	5.46	5.76	0.10	5.88	6.06	0.22
100x300	0.83	1.44	0.42	1.27	1.27	0.38	0.12	1.44	0.45	0.12	1.27	0.53
100x500	2.23	3.05	0.41	1.92	2.78	0.36	2.04	2.74	0.28	2.08	2.81	0.16
100x1000	3.31	3.78	0.25	3.12	3.96	0.21	3.33	3.90	0.17	3.45	3.84	0.14
100x2000	3.53	3.75	0.15	3.44	3.80	0.17	3.32	3.79	0.11	3.46	3.75	0.09
200x600	0.14	0.71	0.08	0.33	0.35	0.06	0.11	0.34	0.05	0.11	0.25	0.05
200x1000	1.52	1.74	0.16	1.18	1.73	0.15	1.01	1.68	0.14	1.32	1.79	0.13
200x2000	3.26	3.79	0.20	3.17	3.89	0.21	3.68	3.90	0.18	3.66	3.96	0.11
200x4000	4.20	4.35	0.25	4.01	4.73	0.16	3.96	4.88	0.13	4.25	4.83	0.09
400x1200	0.47	0.80	0.12	0.32	0.57	0.11	0.32	0.49	0.10	0.27	0.49	0.10
400x2000	2.46	2.46	0.19	1.60	2.17	0.17	1.53	2.24	0.17	2.23	2.33	0.12
400x4000	3.27	3.44	0.13	3.23	3.54	0.13	3.22	3.59	0.12	3.45	3.64	0.12
400x8000	2.87	3.03	0.23	2.83	3.05	0.10	2.84	3.04	0.06	2.91	3.04	0.05

Table 4.4.: Comparison of Fixed Set Search performance in highly connected graphs by varying the number of iterations to find the fixed set (with m_{best} set to 5% of iterations)

Sup x dem	Iterations = 50			Iterations = 100			Iterations = 1000			Iterations = 4000		
	Avg	Max	SD	Avg	Max	SD	Avg	Max	SD	Avg	Max	SD
2x6	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2x10	0.00	3.62	0.28	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2x20	3.77	12.58	1.90	2.20	11.64	1.59	0.00	3.46	0.65	0.00	1.57	0.27
2x40	0.22	4.21	0.90	1.33	3.77	0.92	0.22	2.00	0.80	1.77	1.77	0.80
5x15	1.40	5.14	0.95	0.00	3.74	0.73	0.93	2.34	0.47	0.93	0.93	0.41
5x25	2.83	9.07	1.38	1.13	5.67	1.22	1.13	3.12	0.39	1.13	1.13	0.00
5x50	1.82	6.95	0.93	0.99	4.30	0.87	1.16	4.14	0.88	1.16	3.15	0.83
5x100	0.80	2.06	0.40	0.13	2.12	0.37	0.27	1.46	0.31	0.33	1.33	0.22
10x30	4.81	8.86	1.33	2.03	8.61	1.07	2.03	4.81	0.78	2.03	4.81	0.48
10x50	4.04	7.07	1.10	2.22	7.07	1.02	0.81	3.84	0.58	1.82	2.02	0.51
10x100	0.93	3.49	0.59	0.93	3.02	0.54	0.70	2.71	0.37	1.40	1.40	0.19
10x200	0.24	1.25	0.23	0.91	1.12	0.19	0.34	0.78	0.13	0.20	0.51	0.09
25x75	1.86	10.16	1.31	5.78	9.51	1.33	4.75	7.46	1.29	2.33	5.78	1.05
25x125	3.78	6.34	0.86	2.34	6.79	0.82	1.33	3.11	0.40	2.06	2.56	0.38
25x250	0.86	5.20	0.50	1.60	5.55	0.46	0.71	5.86	0.96	1.29	2.38	0.32
25x500	0.42	1.12	0.16	0.59	1.09	0.15	0.49	0.91	0.12	0.21	0.62	0.09
50x150	4.45	8.13	1.04	5.07	8.13	1.08	5.12	7.27	0.72	5.26	5.93	0.38
50x250	3.49	6.53	0.80	3.60	5.44	0.73	2.53	5.07	0.76	2.69	4.00	0.65
50x500	1.53	2.42	0.29	1.43	2.77	0.29	0.93	2.24	0.27	1.03	1.46	0.17
50x1000	0.45	1.02	0.13	0.55	1.03	0.12	0.62	0.80	0.10	0.49	0.73	0.08
100x300	3.77	7.81	0.74	4.61	6.39	0.57	4.24	5.76	0.39	4.48	5.19	0.29
100x500	4.29	4.59	0.45	3.08	4.48	0.42	2.00	3.26	0.26	2.09	2.74	0.19
100x1000	1.42	2.18	0.19	1.21	1.76	0.18	0.82	1.43	0.16	0.97	1.35	0.14
100x2000	0.46	0.79	0.08	0.46	0.72	0.08	0.40	0.69	0.07	0.40	0.63	0.06
200x600	5.10	7.46	0.62	5.32	6.68	0.53	3.75	5.40	0.38	4.35	5.30	0.28
200x1000	3.67	5.58	0.40	3.92	4.35	0.34	2.78	3.96	0.27	2.67	3.32	0.17
200x2000	1.29	1.90	0.15	1.58	1.95	0.15	1.27	1.72	0.13	1.27	1.53	0.11
200x4000	0.39	1.12	0.07	0.47	0.76	0.06	0.49	0.62	0.05	0.40	0.61	0.05
400x1200	4.99	6.52	0.40	5.13	5.93	0.39	4.07	4.95	0.24	4.37	4.50	0.15
400x2000	3.55	5.35	0.38	3.20	4.62	0.31	3.05	3.81	0.23	3.09	3.66	0.20
400x4000	1.41	1.90	0.12	1.28	1.67	0.10	1.15	1.32	0.06	1.11	1.21	0.05
400x8000	0.48	0.68	0.05	0.48	0.61	0.04	0.45	0.54	0.03	0.43	0.51	0.03

When examining the performance of using a randomized trait for vertex selection (trait 4), as explained in Section 3, we initially assumed that introducing randomness could help explore different regions of the solution space, potentially leading to better overall solutions in cases where deterministic methods might get stuck in local optima.

Table 4.5.: Comparison of Fixed Set Search performance in low connected graphs by varying the number of iterations to find the fixed set (with m_{best} set to 5% of iterations)

Sup x dem	Iterations = 50			Iterations = 100			Iterations = 1000			Iterations = 4000		
	Avg	Max	SD	Avg	Max	SD	Avg	Max	SD	Avg	Max	SD
2x6	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2x10	0.00	5.56	1.40	0.00	5.56	0.35	0.00	0.00	0.00	0.00	0.00	0.00
2x20	2.57	5.47	1.00	2.25	5.47	0.99	1.61	3.22	0.59	1.61	3.22	0.21
2x40	17.69	19.79	4.57	17.86	19.96	5.04	17.69	19.44	5.40	17.69	19.44	5.23
5x15	6.51	13.02	3.20	6.51	13.02	2.92	6.51	13.02	0.65	6.51	6.51	0.00
5x25	6.42	8.94	1.97	0.84	8.94	1.76	0.84	4.47	0.56	0.84	2.23	0.29
5x50	12.58	21.69	3.30	6.56	18.21	2.85	5.35	13.65	2.57	11.78	13.65	2.61
5x100	0.67	3.77	0.62	0.55	3.10	0.51	0.55	2.00	0.23	0.55	1.11	0.13
10x30	4.15	10.14	1.94	2.30	9.91	1.70	4.15	7.83	1.22	4.15	5.53	0.77
10x50	4.27	5.92	1.75	0.69	5.65	1.39	0.69	0.96	0.08	0.69	0.96	0.08
10x100	9.45	16.33	1.85	9.28	15.37	1.68	9.68	10.59	0.27	9.68	9.68	0.00
10x200	3.70	13.06	2.47	7.86	9.69	1.89	3.62	7.95	0.25	3.95	4.20	0.16
25x75	4.74	5.37	0.07	4.83	5.37	0.06	4.74	4.83	0.02	4.74	4.83	0.01
25x125	2.49	6.86	0.71	3.60	6.86	0.46	3.71	4.26	0.12	3.93	3.93	0.09
25x250	5.50	6.42	0.94	4.90	6.32	0.87	2.78	4.65	0.44	3.73	4.02	0.32
25x500	4.38	5.00	0.42	3.89	5.00	0.36	3.82	4.45	0.12	4.02	4.17	0.08
50x150	0.65	2.26	0.31	1.00	2.06	0.23	0.95	1.30	0.10	0.95	1.30	0.03
50x250	3.17	5.10	0.76	1.99	4.65	0.68	1.52	2.79	0.38	1.42	2.41	0.24
50x500	3.33	4.38	0.45	3.20	4.59	0.41	3.14	3.70	0.24	2.85	3.30	0.16
50x1000	5.19	6.60	0.27	5.64	6.39	0.22	5.45	5.70	0.10	5.30	5.61	0.07
100x300	1.81	1.93	0.41	0.71	1.95	0.44	0.66	1.27	0.30	0.12	0.71	0.06
100x500	2.16	3.93	0.54	3.50	3.61	0.46	1.40	2.59	0.33	1.71	2.62	0.32
100x1000	3.54	5.09	0.43	3.42	4.64	0.35	3.34	3.76	0.17	3.17	3.52	0.07
100x2000	3.40	4.81	0.38	3.55	4.78	0.26	3.51	3.73	0.12	3.31	3.75	0.09
200x600	0.20	1.08	0.21	0.66	1.07	0.17	0.20	0.26	0.05	0.11	0.20	0.03
200x1000	1.59	2.50	0.25	1.32	2.09	0.21	1.34	1.65	0.13	1.05	1.57	0.12
200x2000	3.41	4.37	0.27	3.17	3.89	0.21	3.03	3.71	0.19	3.13	3.72	0.19
200x4000	3.69	4.78	0.20	4.01	4.73	0.16	4.10	4.46	0.13	3.96	4.28	0.08
400x1200	0.31	0.92	0.15	0.32	0.57	0.11	0.11	0.52	0.11	0.07	0.43	0.08
400x2000	2.01	2.69	0.23	1.60	2.17	0.17	1.58	2.16	0.17	1.70	2.08	0.15
400x4000	3.05	3.76	0.16	3.07	3.80	0.16	3.18	3.48	0.09	3.08	3.30	0.07
400x8000	3.04	3.30	0.23	2.97	3.17	0.20	2.76	3.03	0.06	2.88	2.98	0.04

However, as shown by the results from Table 4.7, where we compared 500 iterations with 5% of the best solutions (m_{best}) to generate our fixed set, and then applied our Randomized Greedy Algorithm with Pre-Selected Elements (Fixed Set) from Section 3.2.2, the results show that for highly connected graphs, simply performing a final solve using trait 1 with our fixed set (which selects the vertex with the maximum demand that can still be covered) considerably outperforms our randomized approach, even after 1000 iterations with trait 4.

Table 4.6.: Comparison of computation times (in milliseconds) for solving the Fixed Set Search across varying numbers of iterations to generate the fixed set, in both low and highly connected graphs (with trait 1 and m_{best} set to 5% of the total iterations)

Sup x dem	Iterations = 100		Iterations = 500		Iterations = 1000		Iterations = 4000	
	Time in ms		Time in ms		Time in ms		Time in ms	
	high conn.	low conn.	high conn.	low conn.	high conn.	low conn.	high conn.	low conn.
2x6	0.1	0.1	0.7	0.7	0.6	0.6	5.1	2.8
2x10	0.2	0.1	0.9	0.6	1.6	1.2	8.1	4.9
2x20	0.7	0.4	3.8	1.7	7.0	3.6	29.2	13.8
2x40	2.8	1.7	14.6	7.3	27.7	15.7	113.7	60.0
5x15	0.2	0.2	1.0	0.9	1.6	1.8	7.8	7.6
5x25	0.8	0.5	4.2	2.4	7.5	5.2	32.4	21.2
5x50	2.7	0.6	14.5	2.7	27.0	5.7	112.3	23.4
5x100	9.0	0.9	47.7	4.1	88.7	8.6	375.7	33.7
10x30	0.4	0.5	2.4	2.3	4.2	4.8	18.5	18.8
10x50	1.2	1.0	6.7	4.4	12.2	9.3	52.2	37.2
10x100	5.6	1.6	29.2	7.4	55.4	15.5	233.8	61.0
10x200	23.1	3.8	118.9	17.5	229.7	37.2	955.4	144.6
25x75	1.5	1.2	7.8	5.7	15.0	12.2	64.1	47.5
25x125	3.8	2.6	19.5	11.9	37.6	25.5	160.3	95.0
25x250	14.4	3.5	72.7	16.2	142.1	34.7	601.0	128.3
25x500	60.4	20.0	298.8	92.2	594.0	197.3	2476.4	732.1
50x150	4.7	4.3	24.5	20.0	46.5	42.6	197.7	164.2
50x250	8.7	6.4	45.0	30.0	85.3	63.1	364.1	239.0
50x500	33.0	14.6	165.6	67.5	323.4	144.6	1351.1	537.1
50x1000	146.3	59.4	714.8	271.9	1464.3	600.4	5900.0	2199.1
100x300	9.3	10.1	49.7	47.4	90.7	101.1	388.4	372.3
100x500	22.4	15.3	117.3	69.4	223.0	152.6	928.9	556.4
100x1000	77.7	37.6	393.2	170.4	767.3	373.9	3175.9	1371.7
100x2000	323.6	112.7	1618.7	513.1	3209.5	1143.3	12953.9	4100.6
200x600	30.9	30.7	153.9	139.5	303.1	334.9	1193.7	1137.4
200x1000	55.6	38.8	280.2	177.1	549.2	447.2	2216.0	1406.7
200x2000	190.4	91.0	943.3	19755.3	1868.3	1041.9	7739.5	3302.1
200x4000	706.1	242.3	3580.4	1229.2	6977.5	2821.7	28487.2	8794.6
400x1200	93.9	107.0	481.5	530.0	931.3	1037.2	3720.5	3932.3
400x2000	159.9	152.4	826.7	738.1	1571.0	1516.8	6463.6	5631.4
400x4000	467.6	321.1	2403.6	1578.5	4626.3	3258.5	19102.9	12377.4
400x8000	1671.0	661.7	8625.4	3197.2	16740.0	6326.3	67211.1	23683.6

This outcome is based on the fact that when we select a small number of m_{best} solutions from our iterations, the resulting fixed set almost completely covers most of the final MPGSD solution space. Consequently, we are often left with only one or two vertices to be added per subgraph.

In highly connected graphs, when performing trait 1, selecting the vertex with the most available demand that can still be covered by our supply vertex, usually leads to optimal or near-optimal results, as the graph's high connectivity often provides a vertex with demand close to the remaining supply.

Table 4.7.: Comparison of performance and computation time (ms) for varying the amount of randomized greedy iterations (trait 4) after Fixed Set selection across different connectivity levels (with a fixed amount of 500 iterations and m_{best} set to 5% (25) to find the fixed set)

Sup x dem	randomized iterations = 50				randomized iterations = 1000				randomized iterations = 50				randomized iterations = 1000			
	high conn.				high conn.				low conn.				low conn.			
	Avg	Max	SD	Time	Avg	Max	SD	Time	Avg	Max	SD	Time	Avg	Max	SD	Time
2x6	0.00	0.00	0.00	3.2	0.00	0.00	0.00	0.6	0.00	0.00	0.00	0.4	0.00	0.00	0.00	0.6
2x10	0.00	0.00	0.00	3.1	0.00	0.00	0.00	1.3	0.00	0.00	0.00	0.9	0.00	0.00	0.00	1.2
2x20	0.00	1.57	0.29	7.7	0.31	0.63	0.14	8.2	0.00	3.22	1.25	1.9	0.96	3.22	1.17	2.9
2x40	0.00	0.22	0.05	23.3	0.00	0.00	0.00	42.0	19.44	19.44	5.35	8.2	17.69	19.44	5.48	10.6
5x15	0.93	2.34	0.45	1.2	0.93	2.34	0.55	1.5	6.51	13.02	1.55	1.0	6.51	13.02	1.28	1.8
5x25	1.13	1.98	0.64	4.6	1.13	1.70	0.60	7.1	2.51	5.59	0.89	3.1	1.40	4.47	0.81	4.5
5x50	0.66	1.32	0.29	16.0	0.17	0.50	0.10	32.9	4.82	12.32	1.38	3.0	6.02	13.65	1.80	5.3
5x100	0.20	0.73	0.13	53.0	0.13	0.13	0.04	116.1	0.44	2.11	0.38	4.6	1.33	2.11	0.35	7.1
10x30	0.76	4.81	1.26	2.5	4.81	4.81	1.07	4.1	5.53	5.53	1.15	2.6	2.30	5.53	1.19	4.4
10x50	1.41	3.64	0.64	7.4	0.61	2.63	0.49	17.7	0.14	0.69	0.13	5.2	0.14	0.69	0.12	7.5
10x100	1.09	2.09	0.33	32.9	0.62	1.55	0.23	82.1	9.68	10.42	0.63	8.2	9.68	10.42	0.42	12.2
10x200	0.34	0.85	0.14	138.0	0.17	0.41	0.07	370.0	3.24	8.28	0.92	19.9	3.37	7.74	0.45	27.4
25x75	2.89	5.78	1.29	8.5	2.33	5.78	1.31	13.4	4.83	4.83	0.03	6.6	4.74	4.83	0.03	10.0
25x125	3.50	4.06	0.51	22.4	2.28	2.50	0.28	42.9	3.93	4.26	0.20	13.7	3.87	4.26	0.24	19.2
25x250	2.05	2.92	0.32	85.9	1.57	2.00	0.19	214.3	2.12	4.27	0.46	21.1	2.85	4.14	0.46	28.1
25x500	0.71	1.00	0.11	356.1	0.63	0.72	0.07	905.9	4.07	4.54	0.14	104.6	4.07	4.36	0.14	129.0
50x150	5.74	6.84	0.51	27.2	5.45	6.36	0.44	50.1	1.00	1.35	0.11	22.2	0.80	1.20	0.12	32.3
50x250	4.77	5.07	0.42	52.3	3.52	4.27	0.32	107.1	2.15	2.79	0.36	33.0	1.77	2.82	0.36	47.4
50x500	1.92	2.30	0.16	192.4	1.54	1.85	0.13	469.1	2.86	3.41	0.24	74.3	2.40	3.38	0.25	106.9
50x1000	0.96	1.12	0.09	857.4	0.71	0.84	0.06	2330.6	5.66	5.95	0.14	303.1	5.46	5.71	0.11	373.8
100x300	4.58	5.87	0.40	52.2	4.48	5.63	0.36	88.6	0.32	0.71	0.19	50.5	0.12	0.66	0.23	69.7
100x500	4.84	5.19	0.33	128.4	3.73	4.48	0.21	303.0	1.28	2.68	0.38	77.6	2.32	2.48	0.33	109.1
100x1000	2.30	2.62	0.12	449.0	1.93	2.23	0.10	1154.3	2.91	3.82	0.22	191.4	2.67	3.85	0.27	291.7
100x2000	1.09	1.19	0.06	1894.3	0.96	1.00	0.04	5162.7	3.73	4.00	0.17	565.3	3.66	4.03	0.19	703.6
200x600	5.52	6.36	0.42	163.1	4.44	5.92	0.33	285.5	0.12	0.32	0.05	151.3	0.07	0.29	0.05	186.8
200x1000	5.35	5.92	0.27	313.8	4.83	5.39	0.20	724.3	1.63	1.85	0.16	184.7	1.35	1.82	0.17	260.4
200x2000	2.43	2.75	0.10	1092.7	2.24	2.51	0.08	2875.6	3.63	3.96	0.21	441.4	3.37	3.91	0.22	616.1
200x4000	1.16	1.21	0.05	4268.2	1.05	1.09	0.03	11810.3	4.19	5.15	0.17	1201.4	4.30	4.94	0.16	1543.0
400x1200	5.67	6.18	0.22	514.5	5.04	5.91	0.22	886.8	0.40	0.50	0.10	511.2	0.27	0.51	0.11	620.9
400x2000	5.80	6.41	0.21	903.7	5.83	5.93	0.18	2007.2	1.77	2.38	0.16	757.8	1.88	2.23	0.17	997.4
400x4000	2.54	2.89	0.08	2763.0	2.47	2.64	0.05	7063.9	3.42	3.70	0.11	1667.4	3.21	3.63	0.12	2198.3
400x8000	1.15	1.30	0.03	10490.9	1.15	1.17	0.02	27960.0	3.26	3.43	0.14	3160.8	3.17	3.41	0.11	4115.6

The same logic generally applies to low connectivity graphs; however, in some cases, especially with a higher number of iterations, our randomized approach (trait 4) performs slightly better than trait 1. This is because, in low connectivity graphs, selecting vertices only based on the highest demand can occasionally block off other subgraphs. By including traits 2 or 3, which also consider the amount of adjacent vertices, we open up paths that would otherwise have been blocked, resulting in better solutions in some scenarios.

In addition to evaluating the performance, it is crucial to consider the computational efficiency of the FSS approach, as detailed in Table 4.6. The results show that graphs with low connectivity are generally faster to solve compared to highly connected graphs. For instance, solving a 400x8000 problem with 1000 iterations takes 6.3 seconds in a low connected graph, while the same task takes about 16.7 seconds in a highly connected graph. This is due to the fact, that in low connected graphs each vertex, has fewer adjacent vertices, that could be added, so less comparisons between them need to be performed by the algorithm.

Another key observation is that the computation time increases linearly with the number of iterations performed to find the fixed set. For example, when solving the highly connected 10x100 graph, computation time increases from 5.6 milliseconds with 100 iterations to 55.4 milliseconds with 1000 iterations, and further to 233.8 milliseconds with 4000 iterations. This linear growth is consistent across different graph sizes, making it easier to predict the computational time as the number of iterations increases.

When analyzing the time performance after generating the fixed set, we observe that each iteration takes less time than each iteration used to find the fixed set. This is because we no longer need to solve the entire graph again, as we start with a majority of the solution space already covered. As seen in Table 4.7, even with 500 iterations to generate and 1000 iterations after finding the fixed set, we achieve faster results than if we were to perform 1000 iterations solely to find the fixed set. For example, in the case of a 400x8000 low connected graph, the time taken for 500 iterations to find the fixed set, followed by performing 1000 iterations with the randomized approach on our fixed set is approximately 4 seconds, compared to 6 seconds when performing 1000 iterations solely for finding the fixed set, as shown in Table 4.6.

It is also important to note that while we focused on the impact of iterations on computation time, varying the number of m_{best} solutions during testing had little to no impact on overall computation time. Therefore, while iterations greatly influence both accuracy and computation time, adjusting m_{best} can be done with low concern for time complexity.

Overall, the FSS offers a great approach to improve upon traditional greedy methods, especially in low connected graphs, where it not only improves accuracy, but also works particularly well in terms of computational time. Even in highly connected graphs, where plain greedy methods generally work especially well, the FSS can still provide major improvements with the right configuration of m_{best} solutions and iterations. The linear increase in computational time when increasing the amount iterations shows that this method scales predictably, which allows for targeted decisions to be made on balancing accuracy and computational time. This makes the FSS approach a practical and effective choice, especially in scenarios where computational resources must be carefully managed.

However, the randomized approach, using trait 4, is generally less suitable for highly connected graphs, as it tends to be consistently outperformed by trait 1. While trait 4 can offer advantages in some graphs with low connectivity, it only occasionally outperforms the solutions generated by trait 1. In these specific scenarios, the randomness in trait 4 helps to find new solutions that might otherwise be blocked by the deterministic selection of trait 1, yet this benefit is not universal across all cases.

4.4 Comparison with correction methods from the literature

To compare our Fixed Set Search approach with other methods, like the correction method proposed by Jovanovic et al. (2015), we transformed their test instances of general graphs and trees into our JSON format, which was described earlier. After generating these graphs, we applied our method, using 1000 iterations to find the Fixed Set and selecting 25 (2.5%) of these solutions for the Fixed Set. We then performed one greedy iteration using Trait 1 (maximum demand) to generate our final solution. These results are summarized in Table 4.8.

When comparing our results with those using the correction method from the literature (Jovanovic et al. (2015)), as seen in Table A.1 for general graphs and Table A.2 for trees, we see that our approach only surpassed the correction methods in terms of Average Relative Error in a few cases, such as the 2x6 and 2x10 graphs. In terms of Maximum Relative Error, our method outperformed one correction method (the greedy correction) in more than half of the graphs, but it was generally inferior to the other correction methods. Especially, when considering the Multiheuristic correction method, our approach did not outperform it in any of the tested graphs, consistently showing performance approximately 3.4% worse for general graphs and around 7.5% worse for tree graphs.

The following factors may explain why our Fixed Set Search approach underperforms compared to the correction methods used in the literature:

Graph structure and specialization: Jovanovic et al. (2015) designed their correction methods with specific graph structures in mind, such as trees, which allow targeted optimizations. These structured graphs make correction methods, particularly important to exploit predictable connections, like limited branching. Which makes these optimizations especially useful compared to more connected and unstructured graph types. Our approach is applicable to a broad range of graph types, including general graphs, but it lacks these tailored optimizations and therefore struggles in these type of graphs compared to the correction method.

Table 4.8.: Fixed Set Search performance using graphs from the literature, as described by Jovanovic et al. (2015) (1000 iterations, $m_{best} = 2.5\%$, Trait 1 (maximum demand))

Sup x dem	General Graphs				Trees			
	Avg	Max	SD in	Time in s	Avg	Max	SD	Time in s
2x6	1.49	2.34	0.25	0.00	0.00	0.00	0.00	0.00
2x10	1.63	3.09	0.70	0.00	1.44	1.66	0.09	0.00
2x20	1.96	5.31	1.33	0.00	4.76	5.60	0.67	0.00
2x40	1.16	3.44	0.91	0.02	6.22	7.38	0.70	0.01
5x15	5.39	7.87	1.22	0.00	2.59	4.24	0.67	0.00
5x25	4.36	8.27	1.42	0.00	4.84	7.05	1.20	0.00
5x50	2.94	5.41	1.05	0.02	10.22	11.54	1.18	0.01
5x100	1.32	2.97	0.66	0.06	13.64	14.74	1.27	0.03
10x30	6.90	10.08	1.65	0.00	5.16	7.94	1.32	0.00
10x50	5.52	8.50	1.26	0.01	8.33	10.67	1.46	0.01
10x100	3.47	5.30	0.94	0.04	12.31	13.95	1.35	0.02
10x200	1.70	3.34	0.67	0.13	16.22	17.62	0.96	0.06
25x75	8.06	11.20	1.30	0.02	6.62	9.30	1.26	0.01
25x125	7.43	9.37	1.01	0.04	9.77	11.56	1.20	0.02
25x250	3.80	5.37	0.68	0.12	11.45	12.95	0.89	0.06
25x500	2.21	3.40	0.46	0.34	13.42	14.83	0.76	0.17
50x150	9.99	12.05	1.06	0.05	7.16	9.55	1.08	0.03
50x250	7.73	9.44	0.76	0.10	9.47	11.08	0.84	0.06
50x500	4.40	5.42	0.51	0.26	12.15	13.41	0.71	0.15
50x1000	2.84	3.82	0.42	0.77	13.39	14.69	0.65	0.37
100x300	10.66	12.25	0.73	0.11	8.64	10.36	0.84	0.08
100x500	8.32	9.51	0.58	0.22	10.21	11.70	0.67	0.15
100x1000	4.43	5.22	0.37	0.58	12.16	13.36	0.57	0.35
100x2000	2.99	3.72	0.31	1.70	13.23	14.09	0.47	0.95
200x600	11.42	12.46	0.54	0.37	8.79	10.13	0.59	0.29
200x1000	8.52	9.28	0.40	0.65	10.61	11.59	0.52	0.47
200x2000	4.95	5.47	0.28	1.55	12.31	13.17	0.39	0.99
200x4000	3.18	3.64	0.23	4.06	13.66	14.36	0.35	2.36
400x1200	11.80	12.52	0.37	1.10	9.40	10.17	0.41	0.96
400x2000	8.80	9.32	0.29	2.08	10.94	11.50	0.32	1.55
400x4000	4.98	5.42	0.21	4.60	12.51	13.01	0.26	3.40
400x8000	3.08	3.45	0.17	11.13	13.72	14.11	0.23	6.77

Correction method advantages: The correction methods by Jovanovic et al. (2015) include heuristics that iteratively refine the solution, similar to a "hill climbing method" (Jovanovic et al. (2015), p. 392). This process allows for continuous adjustments that improve the solution closer to optimality. In contrast, our Fixed Set Search approach finds a single, fixed set and relies on greedy iterations to finalize the solution, limiting its ability to improve and correct errors. This major difference in methodology is likely why our approach performs worse, particularly when the problem requires continuous fine-tuning, like trees.

Trade-off between speed and accuracy: While our approach shows linear increase in computational time, the correction methods, especially the Multiheuristic correction method, are specifically designed to enhance accuracy, even at the cost of increased computational time. This is clearly visible when examining the graph 400x8000 in Table A.2, where it takes 159.5 seconds to solve the graph using the multiheuristic correction method proposed by Jovanovic et al. (2015). In contrast, our approach is faster compared to the multiheuristic correction method, but it sacrifices some accuracy, making it less effective on graphs where precision is crucial.

The differences between our method and the correction methods by Jovanovic et al. (2015) highlight the importance of tailored heuristics, which iteratively refine solutions to achieve high-quality solutions, especially on structured graph types. Our findings suggest that while our approach lacks the high quality final solutions in certain graphs, it offers computational efficiency and adaptability and provides better initial solutions compared to the basic greedy approach. By integrating our method as an initial solution and then applying existing correction methods, we could enhance overall accuracy, making it a more competitive option for solving MPGSD problems.

5 Conclusion

In this thesis, we presented a method to solve the Maximal Partitioning of Supply and Demand Graphs (MPGSD) using a Fixed Set Search (FSS) approach, leveraging its ability to identify recurring patterns in high-quality solutions to generate a more refined final solution. This approach improves overall solution quality while balancing computational time and accuracy.

Our computational results demonstrated that the proposed approach is well-suited for the MPGSD problem, particularly in low connected graphs where it was consistently able to find near-optimal solutions. Compared to the basic greedy approach, the FSS showed great improvements, highlighting the effectiveness of exploring and utilizing common occurrences of different solutions. The success of this method relies on carefully adjusting parameters, especially increasing the number of iterations to improve accuracy while managing the trade-off with computational time.

However, most of the time, the FSS approach did not match the precision of existing correction methods from the literature, as proposed by Jovanovic et al. (2015). Especially in structured graphs like trees, the inability to refine solutions beyond the initial fixed set represents a key limitation. Nevertheless, the adaptability of the FSS and speed make it a valuable addition to the landscape of heuristic algorithms for graph partitioning.

Our findings suggest that while the FSS provides better initial solutions than the basic greedy approach, integrating it with existing correction methods could enhance overall accuracy and solution quality. By using a high-quality initial solution provided by the FSS followed by applying advanced correction methods to refine the solution, we could combine both methods to utilize the strengths of both methods.

Future work could focus on developing these hybrid methods, combining the FSS with the multiheuristic correction method from the literature to enhance overall solution quality. Additionally, expanding the algorithm by implementing a broad range of greedy traits could lead to a vast exploration of the initial solution space, ultimately resulting in more diverse and higher-quality fixed sets, making the FSS a comprehensive approach in the field of graph partitioning.

For implementation details and the graph instances used, the code and resources are accessible on GitHub: https://github.com/CodingPythagoras/MPGSD-using-FixedSetSearch.git.

A Appendix

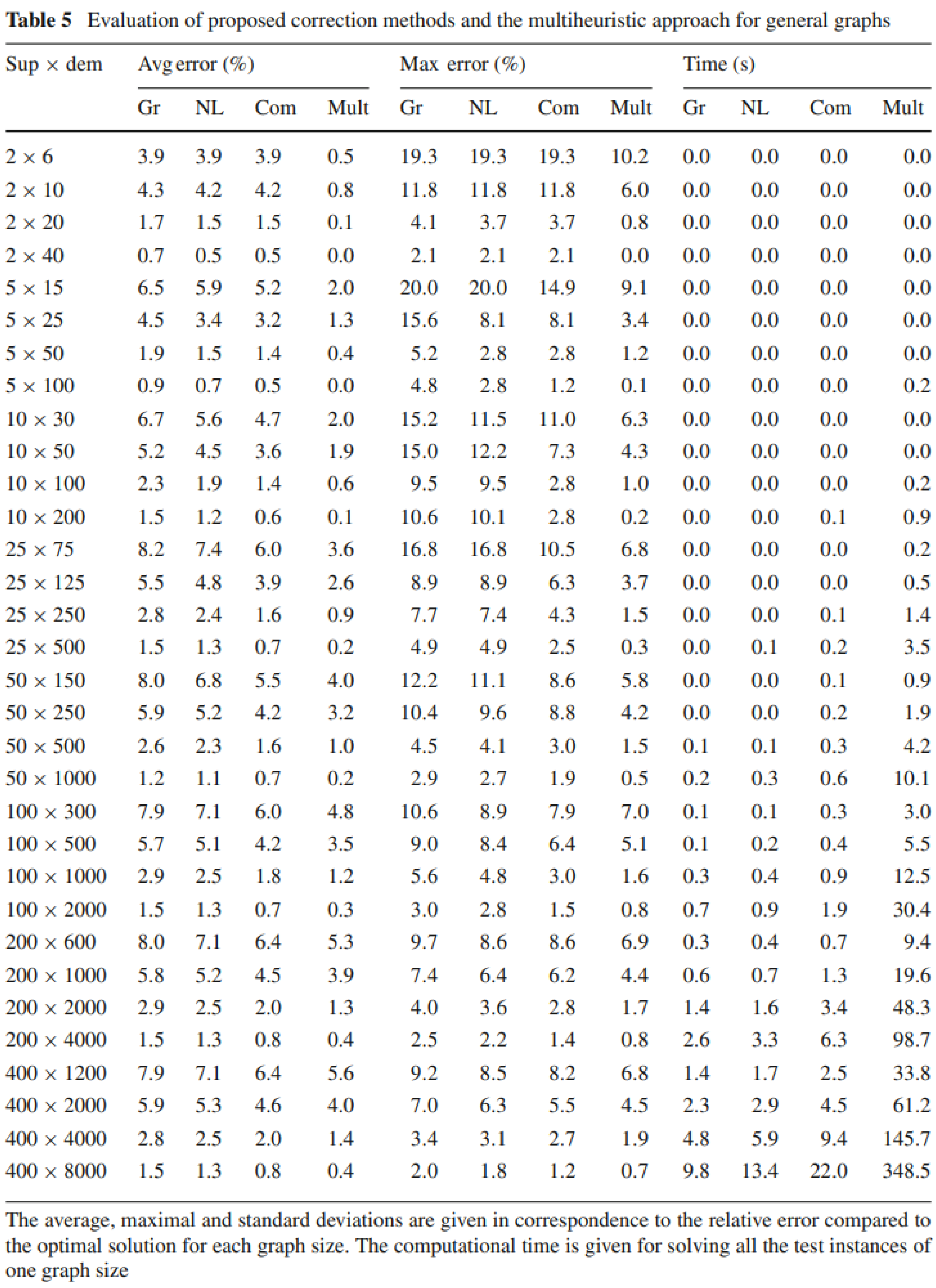

Table 5 Evaluation of proposed correction methods and the multiheuristic approach for general graphs

Sup × dem	Avg error (%)				Max error (%)				Time (s)			
	Gr	NL	Com	Mult	Gr	NL	Com	Mult	Gr	NL	Com	Mult
2 × 6	3.9	3.9	3.9	0.5	19.3	19.3	19.3	10.2	0.0	0.0	0.0	0.0
2 × 10	4.3	4.2	4.2	0.8	11.8	11.8	11.8	6.0	0.0	0.0	0.0	0.0
2 × 20	1.7	1.5	1.5	0.1	4.1	3.7	3.7	0.8	0.0	0.0	0.0	0.0
2 × 40	0.7	0.5	0.5	0.0	2.1	2.1	2.1	0.0	0.0	0.0	0.0	0.0
5 × 15	6.5	5.9	5.2	2.0	20.0	20.0	14.9	9.1	0.0	0.0	0.0	0.0
5 × 25	4.5	3.4	3.2	1.3	15.6	8.1	8.1	3.4	0.0	0.0	0.0	0.0
5 × 50	1.9	1.5	1.4	0.4	5.2	2.8	2.8	1.2	0.0	0.0	0.0	0.0
5 × 100	0.9	0.7	0.5	0.0	4.8	2.8	1.2	0.1	0.0	0.0	0.0	0.2
10 × 30	6.7	5.6	4.7	2.0	15.2	11.5	11.0	6.3	0.0	0.0	0.0	0.0
10 × 50	5.2	4.5	3.6	1.9	15.0	12.2	7.3	4.3	0.0	0.0	0.0	0.0
10 × 100	2.3	1.9	1.4	0.6	9.5	9.5	2.8	1.0	0.0	0.0	0.0	0.2
10 × 200	1.5	1.2	0.6	0.1	10.6	10.1	2.8	0.2	0.0	0.0	0.1	0.9
25 × 75	8.2	7.4	6.0	3.6	16.8	16.8	10.5	6.8	0.0	0.0	0.0	0.2
25 × 125	5.5	4.8	3.9	2.6	8.9	8.9	6.3	3.7	0.0	0.0	0.0	0.5
25 × 250	2.8	2.4	1.6	0.9	7.7	7.4	4.3	1.5	0.0	0.0	0.1	1.4
25 × 500	1.5	1.3	0.7	0.2	4.9	4.9	2.5	0.3	0.0	0.1	0.2	3.5
50 × 150	8.0	6.8	5.5	4.0	12.2	11.1	8.6	5.8	0.0	0.0	0.1	0.9
50 × 250	5.9	5.2	4.2	3.2	10.4	9.6	8.8	4.2	0.0	0.0	0.2	1.9
50 × 500	2.6	2.3	1.6	1.0	4.5	4.1	3.0	1.5	0.1	0.1	0.3	4.2
50 × 1000	1.2	1.1	0.7	0.2	2.9	2.7	1.9	0.5	0.2	0.3	0.6	10.1
100 × 300	7.9	7.1	6.0	4.8	10.6	8.9	7.9	7.0	0.1	0.1	0.3	3.0
100 × 500	5.7	5.1	4.2	3.5	9.0	8.4	6.4	5.1	0.1	0.2	0.4	5.5
100 × 1000	2.9	2.5	1.8	1.2	5.6	4.8	3.0	1.6	0.3	0.4	0.9	12.5
100 × 2000	1.5	1.3	0.7	0.3	3.0	2.8	1.5	0.8	0.7	0.9	1.9	30.4
200 × 600	8.0	7.1	6.4	5.3	9.7	8.6	8.6	6.9	0.3	0.4	0.7	9.4
200 × 1000	5.8	5.2	4.5	3.9	7.4	6.4	6.2	4.4	0.6	0.7	1.3	19.6
200 × 2000	2.9	2.5	2.0	1.3	4.0	3.6	2.8	1.7	1.4	1.6	3.4	48.3
200 × 4000	1.5	1.3	0.8	0.4	2.5	2.2	1.4	0.8	2.6	3.3	6.3	98.7
400 × 1200	7.9	7.1	6.4	5.6	9.2	8.5	8.2	6.8	1.4	1.7	2.5	33.8
400 × 2000	5.9	5.3	4.6	4.0	7.0	6.3	5.5	4.5	2.3	2.9	4.5	61.2
400 × 4000	2.8	2.5	2.0	1.4	3.4	3.1	2.7	1.9	4.8	5.9	9.4	145.7
400 × 8000	1.5	1.3	0.8	0.4	2.0	1.8	1.2	0.7	9.8	13.4	22.0	348.5

The average, maximal and standard deviations are given in correspondence to the relative error compared to the optimal solution for each graph size. The computational time is given for solving all the test instances of one graph size

Figure A.1.: Performance of different correction methods applied to general graphs, as illustrated in Jovanovic et al. (2015), p. 390.

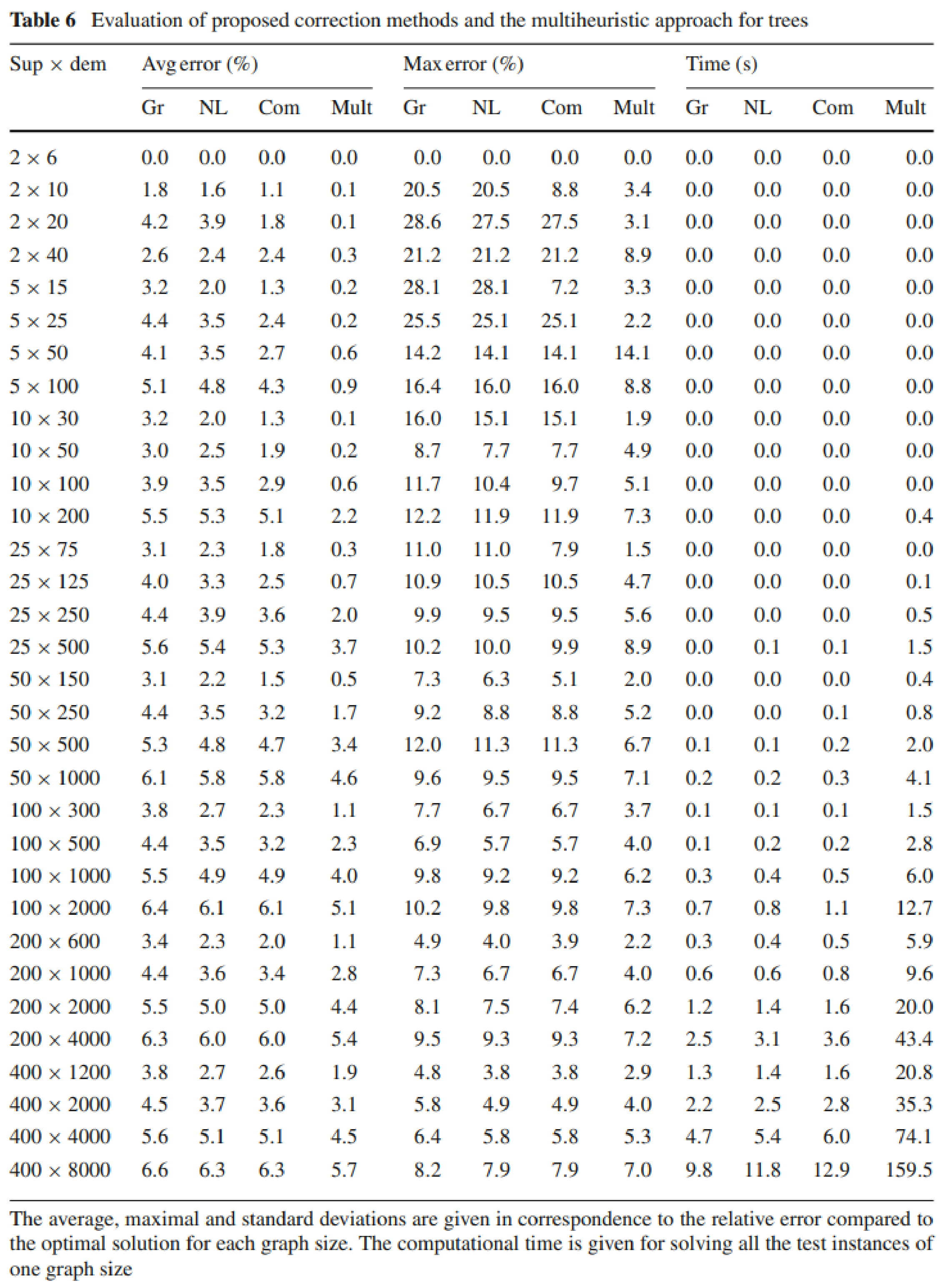

Table 6 Evaluation of proposed correction methods and the multiheuristic approach for trees

Sup × dem	Avg error (%)				Max error (%)				Time (s)			
	Gr	NL	Com	Mult	Gr	NL	Com	Mult	Gr	NL	Com	Mult
2 × 6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2 × 10	1.8	1.6	1.1	0.1	20.5	20.5	8.8	3.4	0.0	0.0	0.0	0.0
2 × 20	4.2	3.9	1.8	0.1	28.6	27.5	27.5	3.1	0.0	0.0	0.0	0.0
2 × 40	2.6	2.4	2.4	0.3	21.2	21.2	21.2	8.9	0.0	0.0	0.0	0.0
5 × 15	3.2	2.0	1.3	0.2	28.1	28.1	7.2	3.3	0.0	0.0	0.0	0.0
5 × 25	4.4	3.5	2.4	0.2	25.5	25.1	25.1	2.2	0.0	0.0	0.0	0.0
5 × 50	4.1	3.5	2.7	0.6	14.2	14.1	14.1	14.1	0.0	0.0	0.0	0.0
5 × 100	5.1	4.8	4.3	0.9	16.4	16.0	16.0	8.8	0.0	0.0	0.0	0.0
10 × 30	3.2	2.0	1.3	0.1	16.0	15.1	15.1	1.9	0.0	0.0	0.0	0.0
10 × 50	3.0	2.5	1.9	0.2	8.7	7.7	7.7	4.9	0.0	0.0	0.0	0.0
10 × 100	3.9	3.5	2.9	0.6	11.7	10.4	9.7	5.1	0.0	0.0	0.0	0.0
10 × 200	5.5	5.3	5.1	2.2	12.2	11.9	11.9	7.3	0.0	0.0	0.0	0.4
25 × 75	3.1	2.3	1.8	0.3	11.0	11.0	7.9	1.5	0.0	0.0	0.0	0.0
25 × 125	4.0	3.3	2.5	0.7	10.9	10.5	10.5	4.7	0.0	0.0	0.0	0.1
25 × 250	4.4	3.9	3.6	2.0	9.9	9.5	9.5	5.6	0.0	0.0	0.0	0.5
25 × 500	5.6	5.4	5.3	3.7	10.2	10.0	9.9	8.9	0.0	0.1	0.1	1.5
50 × 150	3.1	2.2	1.5	0.5	7.3	6.3	5.1	2.0	0.0	0.0	0.0	0.4
50 × 250	4.4	3.5	3.2	1.7	9.2	8.8	8.8	5.2	0.0	0.0	0.1	0.8
50 × 500	5.3	4.8	4.7	3.4	12.0	11.3	11.3	6.7	0.1	0.1	0.2	2.0
50 × 1000	6.1	5.8	5.8	4.6	9.6	9.5	9.5	7.1	0.2	0.2	0.3	4.1
100 × 300	3.8	2.7	2.3	1.1	7.7	6.7	6.7	3.7	0.1	0.1	0.1	1.5
100 × 500	4.4	3.5	3.2	2.3	6.9	5.7	5.7	4.0	0.1	0.2	0.2	2.8
100 × 1000	5.5	4.9	4.9	4.0	9.8	9.2	9.2	6.2	0.3	0.4	0.5	6.0
100 × 2000	6.4	6.1	6.1	5.1	10.2	9.8	9.8	7.3	0.7	0.8	1.1	12.7
200 × 600	3.4	2.3	2.0	1.1	4.9	4.0	3.9	2.2	0.3	0.4	0.5	5.9
200 × 1000	4.4	3.6	3.4	2.8	7.3	6.7	6.7	4.0	0.6	0.6	0.8	9.6
200 × 2000	5.5	5.0	5.0	4.4	8.1	7.5	7.4	6.2	1.2	1.4	1.6	20.0
200 × 4000	6.3	6.0	6.0	5.4	9.5	9.3	9.3	7.2	2.5	3.1	3.6	43.4
400 × 1200	3.8	2.7	2.6	1.9	4.8	3.8	3.8	2.9	1.3	1.4	1.6	20.8
400 × 2000	4.5	3.7	3.6	3.1	5.8	4.9	4.9	4.0	2.2	2.5	2.8	35.3
400 × 4000	5.6	5.1	5.1	4.5	6.4	5.8	5.8	5.3	4.7	5.4	6.0	74.1
400 × 8000	6.6	6.3	6.3	5.7	8.2	7.9	7.9	7.0	9.8	11.8	12.9	159.5

The average, maximal and standard deviations are given in correspondence to the relative error compared to the optimal solution for each graph size. The computational time is given for solving all the test instances of one graph size

Figure A.2.: Performance of different correction methods applied to trees, as illustrated in Jovanovic et al. (2015), p. 391.

Bibliography

Bertsimas, D. and J. N. Tsitsiklis (1997). *Introduction to linear optimization*, Volume 6. Athena Scientific Belmont, MA.

Buluç, A., H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz (2016). *Recent advances in graph partitioning*. Springer.

Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein (2022). *Introduction to algorithms*. MIT press.

Dorigo, M., M. Birattari, and T. Stutzle (2006). Ant colony optimization. *IEEE computational intelligence magazine 1*(4), 28–39.

Fan, W., M. Liu, C. Tian, R. Xu, and J. Zhou (2020). Incrementalization of graph partitioning algorithms. *Proceedings of the VLDB Endowment 13*(8), 1261–1274.

Ito, T., E. D. Demaine, X. Zhou, and T. Nishizeki (2008). Approximability of partitioning graphs with supply and demand. *Journal of Discrete Algorithms 6*(4), 627–650.

Ito, T., X. Zhou, and T. Nishizeki (2009). Partitioning graphs of supply and demand. *Discrete Applied Mathematics 157*(12), 2620–2633.

Jovanovic, R., A. Bousselham, and S. Voß (2015). A heuristic method for solving the problem of partitioning graphs with supply and demand. *Annals of Operations Research 235*, 371–393.

Jovanovic, R., A. Bousselham, and S. Voß (2018). Partitioning of supply/demand graphs with capacity limitations: an ant colony approach. *Journal of Combinatorial Optimization 35*, 224–249.

Jovanovic, R., A. P. Sanfilippo, and S. Voß (2023). Fixed set search applied to the clique partitioning problem. *European Journal of Operational Research 309*(1), 65–81.

Jovanovic, R., M. Tuba, and S. Voss (2016). An ant colony optimization algorithm for partitioning graphs with supply and demand. *Applied Soft Computing 41*, 317–330.

Jovanovic, R., M. Tuba, and S. Voß (2019). Fixed set search applied to the traveling salesman problem. In *Hybrid Metaheuristics: 11th International Workshop, HM 2019, Concepción, Chile, January 16–18, 2019, Proceedings 11*. Springer, 63–77.

Jovanovic, R. and S. Voß (2016). A mixed integer program for partitioning graphs with supply and demand emphasizing sparse graphs. *Optimization Letters 10*, 1693–1703.

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang Information Systems selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Hamburg, den ____01.10.2024_____ Unterschri