Evolutionary Computation
Practical Assignment 2 -
MLS, ILS, and GLS for Graph Bipartitioning

Date: 26/03/2025
Name: Cem Kaya – 9276866
Name: Mert Suoglu - 7331797
Programming Language: Python 3.11
CPU Specifications: AMD R9 7945hx3d, Apple M3 PRO

1. Introduction

In this assignment we implemented the Fiduccia-Mattheyses (FM) heuristic as for local search and combined it with Multi-start Local Search (MLS), Incremental Local Search (ILS) and Genetic Local Search (GLS) algorithms to solve a graph bipartitioning problem for the given graph for the assignment. The aim is to investigate the behavior of meta-heuristic algorithms through experimentation and compare their performances to MLS and to each other. In addition, we investigated 2 more strategies. We implemented an adaptive version of ILS (ILS-ADA) using the Probability Matching algorithm presented in Lecture 5. Lastly, we compared the performance of Steady State Local Search (SSGLS) to the Generational Genetic Local Search (GGLS), where more offsprings are created instead of one.

Report outline

In Section 2, we will give a brief overview about implementation of algorithms. The methodology is in Section 3. The experiments involving MLS, ILS, GLS and additional experiments ILS-ADA and SSGLS vs. GGLS are discussed in respective subsections of Section 4. The overall results are discussed in Section 5 and in conclusion.

2. Implementation

We used python 3.11 as programming language. Even though there are more performant alternatives to Python (e.g. C++) for intensive computation, considering the size of the task, we preferred Python for convenience reasons and to speed up the implementation phase.
The FM local search is implemented as a separate class "FM". We implemented 2 gain buckets, which are basically 2 integer arrays, to find the vertex with maximum gain, move it to other partition and update the neighbor vertices in constant time. Each bucket item serves as a pointer to a linked list of vertices, as it was described in the course. We implemented additional check to ensure that generated partitions are in equal size. For details, please refer to the source code in "*fm_impl.py*" file. Running a full FM local search for 500 vertices takes around 0.04 seconds. For an overview of relevant code files, please refer to the Appendix 1.

We used a test-driven-development approach to make sure that the fundamental functionality works reliable. The unit and functionality tests can be found in 'test_methods_1.py' file. Please see the Appendix 2 for available test cases.

3. Methodology

We implemented MLS, ILS and GLS algorithms as instructed and compared their results after each algorithm was run 10 times with 10.000 FM passes. We found that ILS is the best performing algorithm among all. The comparison of these 3 algorithms is given in section 4. For ILS, we also recorded the number of cases, where ILS stuck in in local optimum for ILS experiments and compared these between different mutation sizes. These are the cases, where 2 consequent solutions have the same cut size in ILS. Such a case will be referred as an 'ILS-Trap' in this report. To find a good mutation size for ILS, we designed a search process. More information about this process is provided in subsection ILS in the section 4. To compare all 3 algorithms and our own investigation topics, we conducted 2 experiments, first we run all algorithms once for 10.000 FM passes. Second, all algorithms were run 25 times with a fixed CPU time, 450 seconds each. 450 seconds was the time required for a single run of 10.000 FM passes.
For statistical significance tests, the Wilcoxon-Mann-Whitney implementation from 'scipy.stats' python package was used, with alpha = 0.05.

Disclaimer regarding usage of large language models:
We used Microsoft Copilot with Anthrophic's Claude 3.5 Sonnet LLM model embedded in Visual Studio Code environment. LLM was used mostly for boiler plate code, such as generating plots for report, loading datasets from archive files, formatting datasets for analysis or drawing plots and charts. We avoided using LLM for the development of core functionality such as FM local search algorithm. The LLM generated code mostly needed further modification. We marked the code locations where we used LLM with a special comment phrase like '`#LLM Prompt: convert the dataframe to html table.` '. It is not feasible to provide all the prompts and LLM outputs in this report, but we provided various samples in Appendix 3.

4. Experiments

In this section we will explain how each experiment is designed and how we tuned the algorithm parameters, as well as discuss our findings and compare the performance of algorithms. Overall results are given in table 1. The computation time is the mean of 10 runs in seconds. The mutation size for ILS is 70, and GLS population size was fixed to 50.

	MLS	**ILS**	**GLS**	**Statistical Significance Test Results**			
Best Cut Size	23	**2**	5		MLS	ILS	GLS
Mean	27.8	8.8	8.0	MLS	1.0000	**0.0002**	**0.0002**
Stdev	2.394	3.08	1.96	ILS	-	1.0000	0.2492
Computation Time – Mean	439.552	444.173	425.63	GLS	-	-	1.0000

Table 1: Result summary

As show in Table 1, ILS is the best performing algorithm with cut size 2. It is worth noting that the cut size 2 is possibly the global optimum. A full visualization of this graph can be found in Appendix 4. However, the performance advantage of ILS over GLS is statistically not significant. As visualized in Figure 1, ILS and GLS distributions are overlapping, while MLS stands out. Clearly, MLS is the worst performing algorithm, which is also supported by significance test results in Table 1 above.

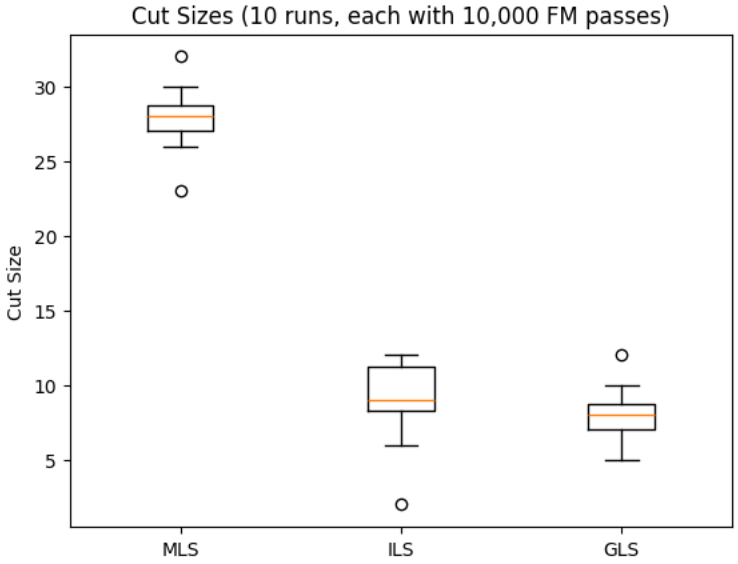


Figure 1: Algorithm performance comparison

4.1. MLS

The MLS starts a new FM local search for a randomly generated solution. Each run contains 10.000 FM passes. 10 MLS runs took 4395 seconds (appx. 73 minutes) on Apple M3 PRO CPU in total. The execution metrics of 10 MLS runs are given provided in Figure 2 below.

Best Cut	Time Elapsed	Avg Time Per FM
29	436.805	0.044
28	437.635	0.044
23	**438.189**	**0.044**
27	439.332	0.044
26	440.498	0.044
32	439.297	0.044
28	440.802	0.044
30	443.679	0.044
27	441.501	0.044
28	437.781	0.044

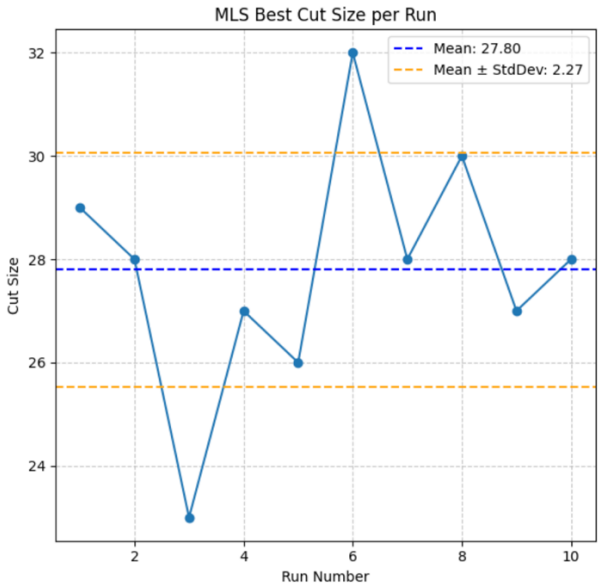


Figure 2: MLS results for 10 runs

4.2. ILS

The Iterative Local Search explores the search space incrementally by mutating an existing solution. If a better solution is found, it replaces the old solution, and it will be mutated in the next iteration. If the solution is worse, it is reverted. Our mutation operator simply swaps the partitioning of a random set of vertices. The size of the random set is determined by *mutation size* parameter. The performance of ILS depends on finding a good value for this parameter. Therefore, we tested different mutation sizes iteratively. We started with size 10 and kept incrementing the mutation size by another 10 and kept the algorithm running until 2 consequent mutation sizes delivered worse results. We found that the **average best cut at 60**, and **absolute best cut at 70** as shown in the Figure 3.

Figure 3: Relation between mutation size and best cut size

As we can observe in Figure 3, a good candidate is the **mutation size 60, with average of 7.2 and best cut size is 3**. The intersection of the orange line ILS-Traps[1] with the cut size line suggests that mutation size 60 provides a good trade-off between inheritance and exploration of new local spaces. However, **next best candidate 70 found a better absolute cut size of 2 with a slight increase in average cut size 8.8**.

Mutation Size: 60, Avg: 7.2			**Mutation Size: 70, Avg: 8.8**		
Best Cut	Time Elapsed	Stucks in Local Optimum	Best Cut	Time Elapsed	Stucks in Local Optimum
8	444.252	109	**2**	**446.810**	**17**
3	**434.504**	**143**	9	437.065	20
5	445.956	127	9	445.580	12
7	434.900	96	8	442.682	10
13	441.297	113	9	446.642	33
7	435.281	95	9	442.561	13
8	442.229	92	12	445.427	9
5	435.860	94	6	444.544	16

[1] ILS-traps are the cases, where ILS found a solution at generation 't' and it has the same cut size with the previous generation 't-1', meaning that ILS stayed in the local optimum.

9	444.304	117	12	446.317	12
7	436.920	101	12	444.104	17

Table 2: Comparison of best mutation sizes

The significance test p-value = 0.116 suggests that difference between 2 cut size averages is not significant. Thus, we decided to use 70 as mutation size for the rest of the experimentation, because it has the better cut value of 2 and it has much lower number of ILS-Traps than size 60 meaning with the rate of 70, ILS has more chance to discover other neighboring search spaces.

The relation between ILS performance, mutation size and number of ILS-Traps can be observed in Figure 3. First observation is that the number of ILS-Traps decreases as the mutation size increases, because higher disruption rate introduces more diversity and increase the chances of finding new neighborhoods. This effect increases the performance of ILS in the beginning. The lower perturbation rates until 50 prevent ILS from discovering better solutions. In region [50-70], the search process finds a better balance between inheriting good solution parts but still manage to keep some diversity in the population enough to explore new neighborhoods. After point 70, the higher perturbation rates become too disruptive, and the performance of ILS drops. To investigate the effect of mutation size parameter, we compared the performance of MLS and ILS with various perturbation sizes. In the table 4, we provide the average cut sizes and p-values. ILS performed better than MLS for all mutation sizes. However, the difference is significant only after mutation size 30. ILS with mutation size below 30 behaves like MLS, in terms of exploring new local spaces. We can observe that the performance difference of average cut sizes is not statistically significant for mutation sizes between [50-70]. The values outside this region delivered significantly worse results, suggesting the optimum mutation size lies in this region.

	Cut (Avg)	*ILS-10*	*ILS-20*	*ILS-30*	*ILS-40*	*ILS-50*	*ILS-56*	*ILS-60*	*ILS-70*	*ILS-80*	*ILS-90*
MLS	27.8	0.3053	0.1201	**0.0002**	**0.0002**	**0.0002**	**0.0002**	**0.0002**	**0.0002**	**0.0002**	**0.0002**
ILS-10	27	1.0000	0.3423	**0.0003**	**0.0002**	**0.0002**	**0.0002**	**0.0002**	**0.0002**	**0.0002**	**0.0005**
ILS-20	23.2	-	1.0000	**0.0063**	**0.0032**	**0.0002**	**0.0003**	**0.0002**	**0.0004**	**0.0011**	**0.0166**
ILS-30	14.1	-	-	1.0000	0.4469	**0.0011**	**0.0034**	**0.0006**	**0.0009**	0.0999	0.5374
ILS-40	12.8	-	-	-	1.0000	**0.0134**	0.0625	**0.0078**	0.0520	0.5949	**0.2235**
ILS-50	7.6	-	-	-	-	1.0000	0.2187	0.8171	0.2147	**0.0146**	**0.0008**
ILS-56	9.1	-	-	-	-	-	1.0000	0.0925	0.9693	0.0799	**0.0021**
ILS-60	7.2	-	-	-	-	-	-	1.0000	0.1166	**0.0078**	**0.0005**
ILS-70	8.8	-	-	-	-	-	-	-	1.0000	**0.0419**	**0.0004**
ILS-80	11.5	-	-	-	-	-	-	-	-	1.0000	**0.0353**
ILS-90	15	-	-	-	-	-	-	-	-	-	1.0000

Table 3: Comparison of MLS and ILS with different perturbation sizes

4.3. GLS

In our GLS implementation, we use a steady state Genetic Local Search algorithm that only produces one child per "generation". This steady state GA uses FM as the local search The population size was fixed to 50 when running the algorithm as instructed. At every iteration two parents are chosen at random, and a new balanced child is produced using a customized uniform crossover. This uniform crossover first looks at the L1 norm (equal to hamming distance for

binary vectors). If the hamming distance is higher than ℓ /2 the labels of the solution in the second parent are flipped. This custom uniform crossover also ensures that the output is balanced. Afterwards this child is optimized using FM and competes with the worst member of the population. Either the child or the worst member is eliminated depending on the result of their minimum cut size.

Best Cut	Time Elapsed
9	421.7578
6	422.7622
7	427.6440
10	427.7142
5	425.9172
8	426.5975
8	426.0488
12	426.2514
7	424.6135
8	426.9743

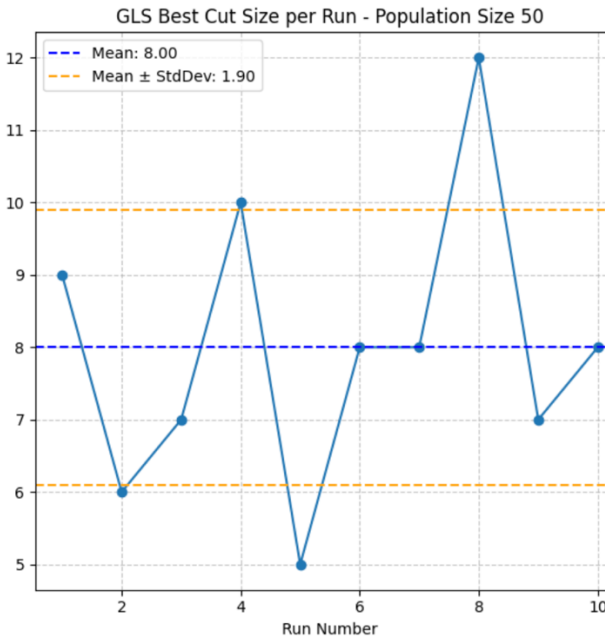


Figure 4: GLS results for 10 runs

4.4. Adaptive ILS (ILS-ADA)

We decided to investigate the effects of dynamically adapting the mutation size (operator) based on gained rewards in ILS algorithm. Thus, we extended our ILS implementation by integrating the probability matching algorithm from the lecture 5.

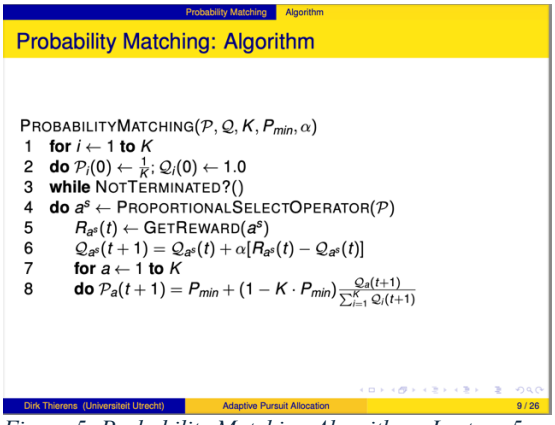


Figure 5: Probability Matching Algorithm - Lecture 5

Based on the results of parameter search, found that the combination of $\mathbf{P_{min}=0.001}$, $\boldsymbol{\alpha = 0.1}$ has the lowest cut 5 and average cut size 16.

We preferred probability matching over pursuit method, because we have a stationary environment. The partitioning changes but, edges and vertices in the graph do not change. To investigate effect of P_{min} and α values, we performed a parameter search. We tested adaptive ILS with various minimum probabilities and alpha values. In total 16 parameter combinations were tested with 10 runs x 2000 FM passes for each. The parameter search configuration is given in figure 6.

```
K = [30,35,40,45,50,55,60,65,70,75,80,85]
iterations = 2000
p_mins = [0.04,0.01,0.001,0.0001]
alphas = [0.1,0.2,0.4,0.6]
```

Figure 6: ILS-ADA Parameter Search Configuration

The search results for all parameter combinations are given in Figure 7.

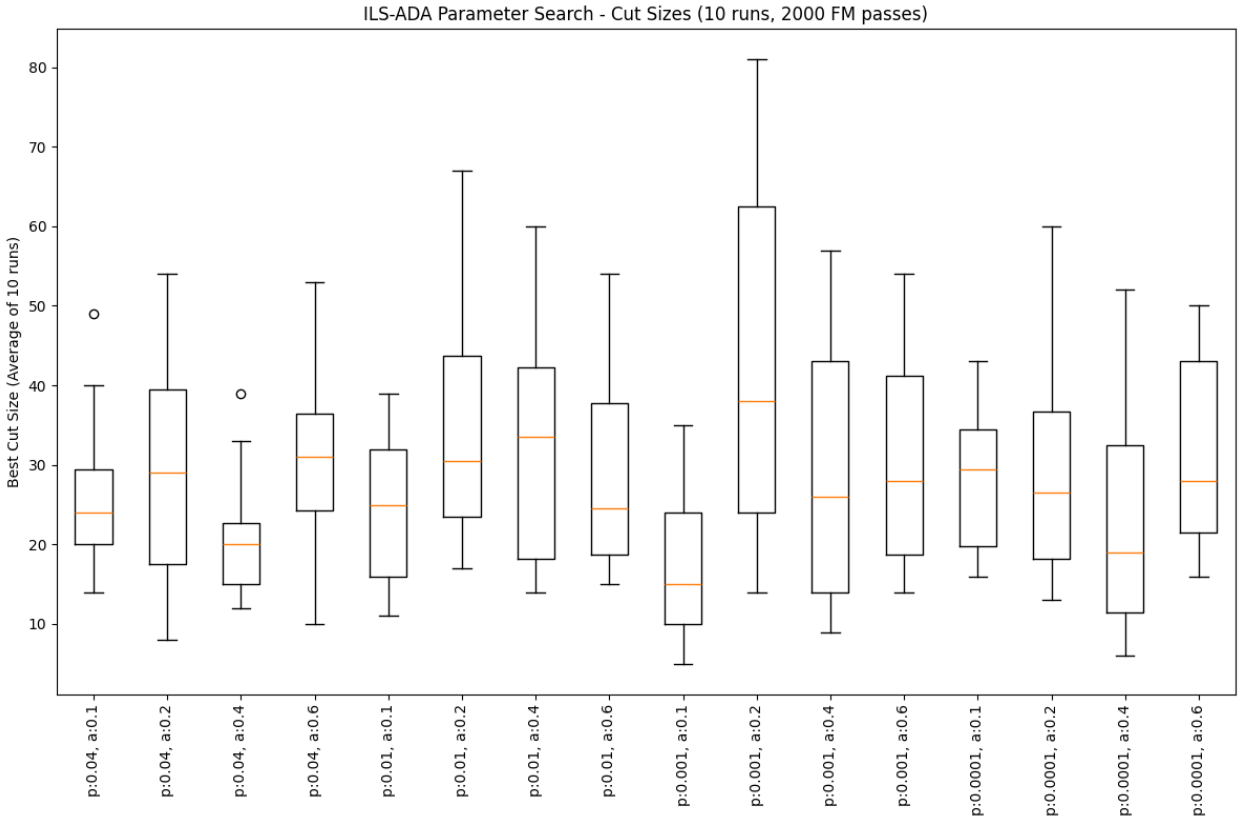


Figure 7: Adaptive ILS parameter search results

We tested the performance of ILS-ADA once for 10.000 FM runs and 25 times with fixed CPU time of 450 seconds. We found **best cut value 3** with mean of 10.12 for 25 runs and a **best cut value of 8** for single run with 10.000 FM passes. The best cuts tend to converge around mutation size 55, as shown in the Figure 8. The comparison to other algorithms and significance tests will be discussed in the upcoming sections.

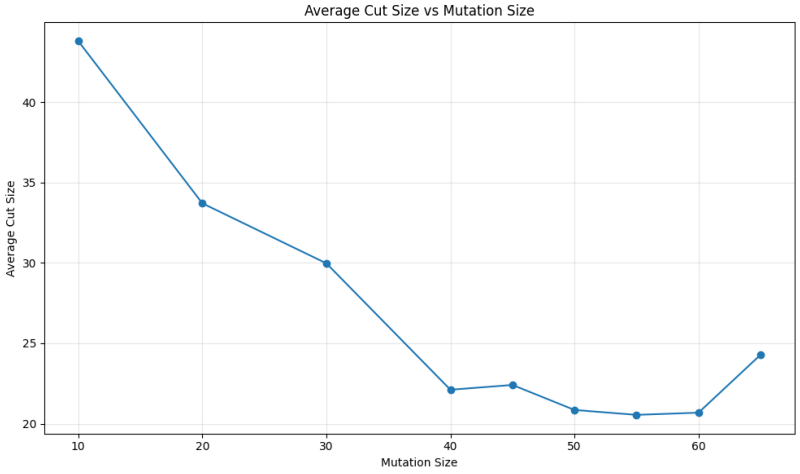


Figure 8: ILS-ADA mutation operator convergence

4.5. Stady State Genetic Algorithm vs Generational Genetic Algorithm

As secondary research question we decided to investigate how does the performance of the steady state genetic algorithm compare to the generational genetic algorithm for the graph bipartitioning when both are allowed to perform the same number of FM passes (1_000) with the same number of population size? How does their relative performance and computational requirements change? Do they converge at different iteration counts? How does changing the population size effect the behavior and efficiency of these algorithms.
As our SSGLS we have used the GLS in Section 4.3. We have implemented a generational GLS class named GLS_GEN for our GGLS. The main difference between SSGLS and GGLS is that each "generation" the GGLS randomly matches all the members of the population in pairs and produces N/2 children. These, children are further optimized using FM. Afterwards top N of the population is chosen as the next generation.

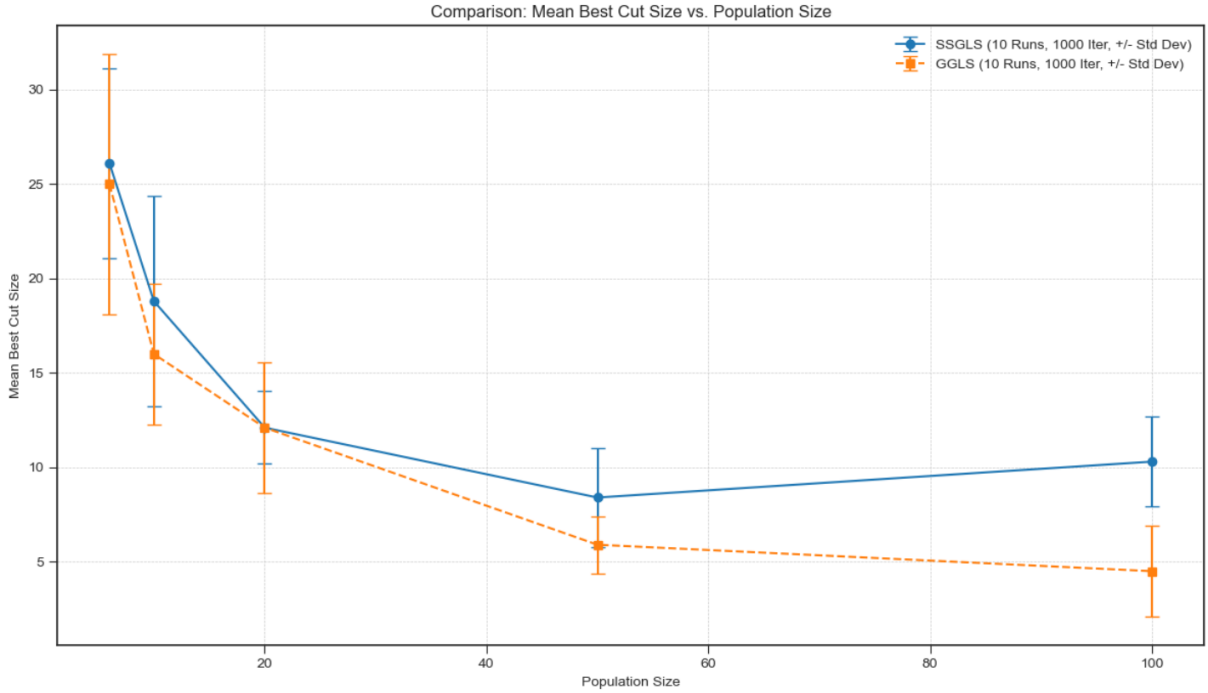


Figure 9: GGLS VS SSGLS for different population sizes

This plot visualizes the comparison between GGLS and SSGLS algorithms based on their output cut size performance across varying population sizes. For larger population sizes the results are statistically different. The GGLS algorithm performs significantly better compared to SSGLS.

Pop Size	Runs	Mean (SS/GG)	Median (SS/GG)	p-val (SS<GG)	p-val (SS>GG)	Significant
6	1000	26.10 / 25.00	25.50 / 28.00	0.6616	0.3665	No
10	1000	18.80 / 16.00	19.50 / 15.00	0.9021	0.1117	No
20	1000	12.10 / 12.10	12.50 / 13.00	0.3656	0.6627	No

50	1000	8.40 / 5.90	8.50 / 6.50	0.987	0.0158	Yes
100	1000	10.30 / 4.50	11.00 / 4.00	0.9996	0.0005	Yes

Figure 10: Statistical test result (Alpha set to 0.05)

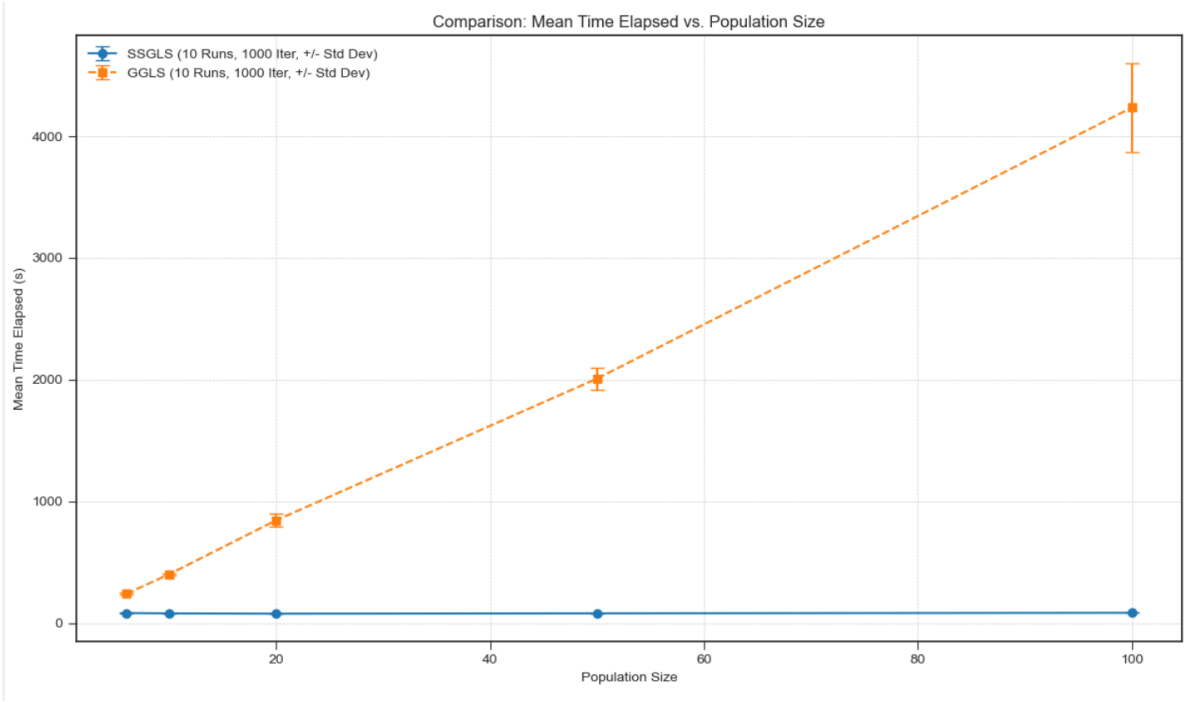


Figure 11: SSGLSVS GGLS wall clock

GGLS was significantly more computationally expensive than SSGLS when run for the same number of FM passes.

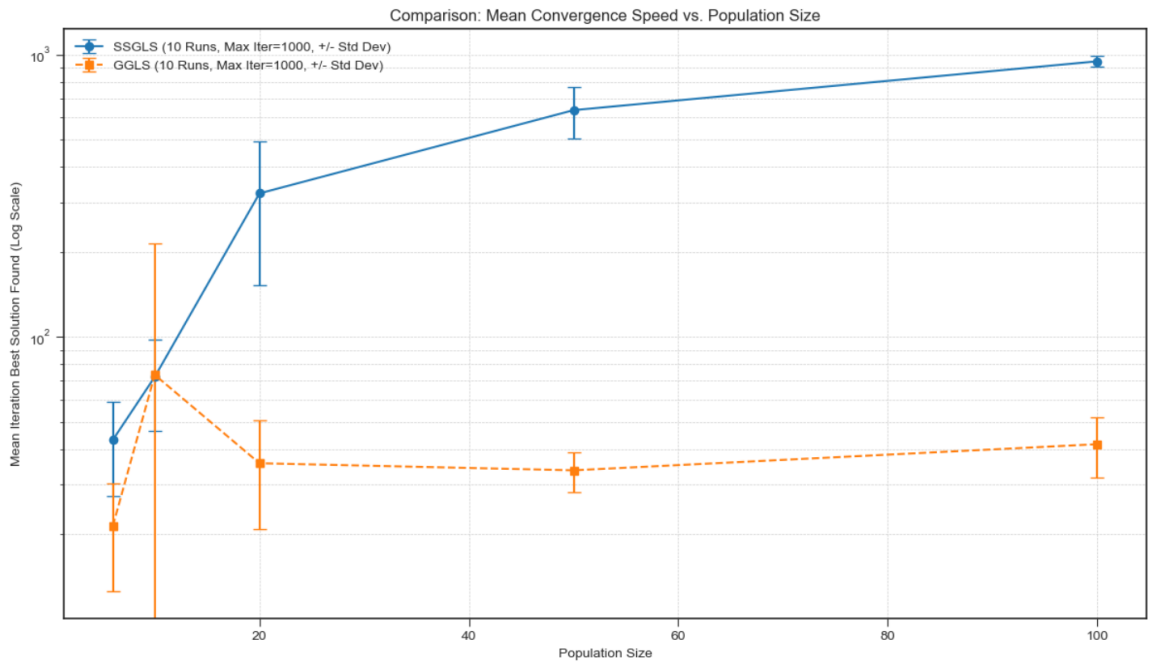


Figure 12: SSGLSVS GGLS Required convergence iteration

GGLS converged significantly earlier compared to SSGLS rendering most of the FM passes at the end irrelevant. This effect was statistically significant.

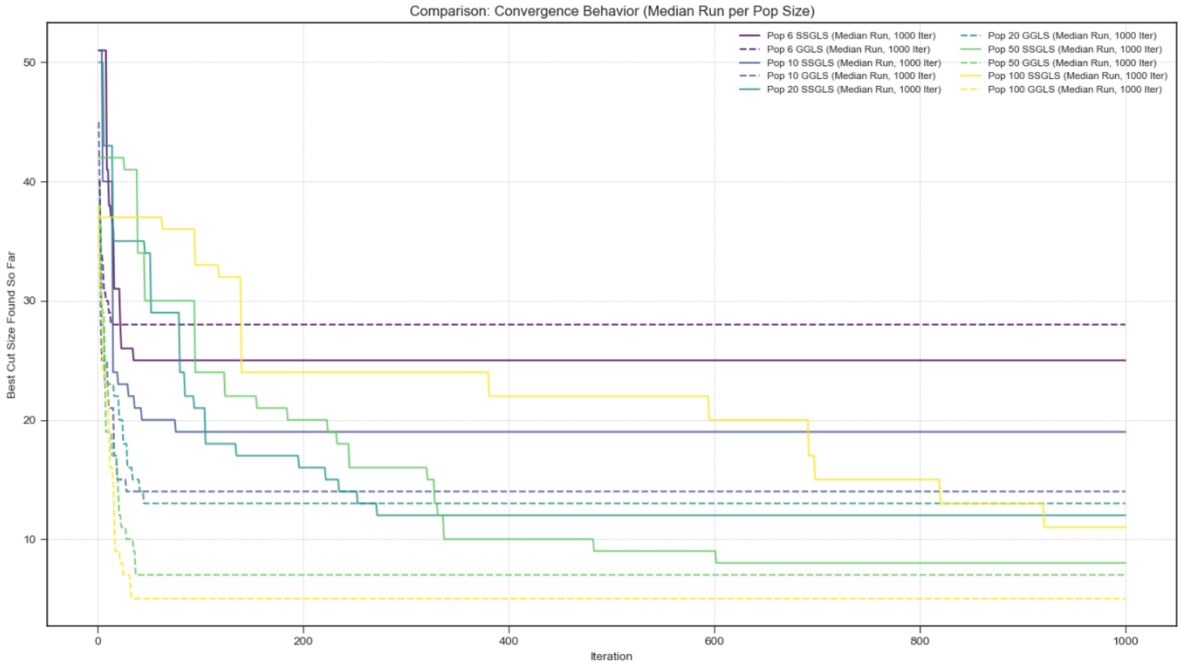


Figure 13: SSGLSVS GGLS Convergence plot

This plot shows a comparison of convergence behavior between SSGLS and GGLS algorithms across different population sizes. GGLS converges faster and to a lower cut size

compared to the SSGLS. Meanwhile SSGLS shows delayed and smoother improvements. SSGLS performed best at population size 50, where GGLS performs best at population size 100.

Pop size	Num runs	Mean best cut size	Std dev best cut size	Min best cut size	Max best cut size	Mean time elapsed	Std dev time elapsed	Mean convergence iteration
6	10	26.1	5.01	19	36	82.87	5.52	43.2
10	10	18.8	5.56	10	26	80.03	2.56	72.2
20	10	12.1	1.92	9	15	77.78	0.54	322.3
50	10	8.4	2.62	3	12	80.37	0.71	635.4
100	10	10.3	2.37	6	14	84.9	0.5	946.2

Figure 14: SSGLS table

Pop size	Num runs	Mean best cut size	Std dev best cut size	Min best cut size	Max best cut size	Mean time elapsed	Std dev time elapsed	Mean convergence iteration
6	10	25	6.88	15	34	240.97	7.66	21.4
10	10	16	3.71	10	23	401.37	10.04	73.4
20	10	12.1	3.48	6	16	844.57	53.61	35.7
50	10	5.9	1.51	3	8	2008.29	89.99	33.7
100	10	4.5	2.42	2	9	4234.12	364.97	41.7

Figure 15: GGLS table

5. Results

This section contains the final comparison of all algorithms to each other. The algorithms are compared in 2 different ways; first we performed a single run with 10000 FM passes for each. Second, each algorithm was run 25 times for a fixed CPU time of 450 seconds. The fixed time is the approximate computational time required for single run with 10.000 FM passes for MLS rounded up.

Algorithm	Best Cut	Time (s)
GGLS (pop=50)	**8**	10302.390
ILS-ADA	**8**	439.570
ILS	9	449.797
SSGLS	9	437.171
MLS	26	469.947

Table 4: Results of 10.000 FM passes

Table 4 shows that the best-performing algorithms are GGLS and ILS-ADA, followed by ILS and SSGLS. All algorithms are significantly better than MLS. The results of single run with 10.000 FM passes are also reflected in second experiment, 25 runs with fixed CPU time, as displayed in Figure 9.

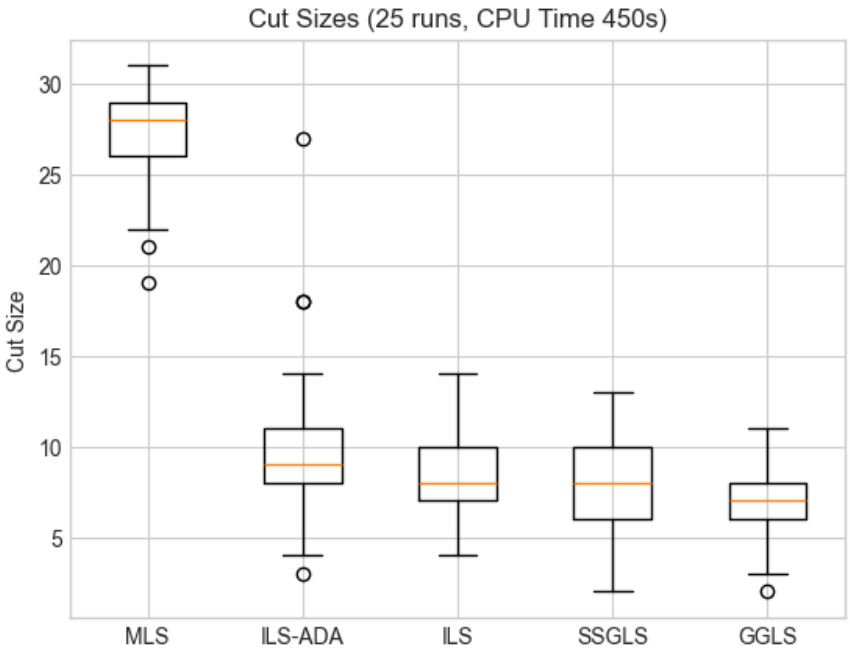


Figure 9: Algorithm comparison - cut size distributions

The Table 5 shows that MLS is the worst performing algorithm as expected, because it just starts a new local search with a random solution. Other algorithms have close performance metrics, which is also supported by the box plot in Figure 9, where all distributions overlap closely, except MLS.

Algorithm	Best Cut	Mean	Median	Std Dev
MLS	19	27.16	28.00	3.06
ILS	4	8.92	8.00	2.58
ILS-ADA	3	10.12	9.00	4.91
GLS	2	8.00	8.00	2.43
GGLS	2	6.72	7.00	2.09

Table 5: Results of 25 runs of fixed CPU time

The significance tests indicate that ILS, SSGLS, and GGLS perform significantly better than the other algorithms tested. Among these, GGLS stands out as analyzed in more detail in Section 4.5, where it is shown to outperform SSGLS as the population size increases. In the current time-limited setting with a population size of 50, GGLS is not yet statistically significantly better than SSGLS, but it is very close to the threshold, and it is better than the ILS result where SSGLS is not better than the ILS.

	MLS	ILS-ADA	ILS	SSGLS	GGLS
MLS	1	**~0.0**	**~0.0**	**~0.0**	**~0.0**
ILS-ADA		1	0.462	0.132	**0.001**

ILS			1	0.327	**0.003**
SSGLS				1	0.058
GGLS					1

Table 6: Significance test results - algorithm performance

6. Conclusion

In this assignment we investigated MLS and various meta-heuristic algorithms and compared their performances through several experiments. We also investigated 2 different strategies, including adaptive ILS and GGLS.

We also found that adaptive ILS performs as efficient as regular ILS. The advantage of adaptive ILS is however, is learning a good mutation rate in progress, thus saving the effort to search for a good mutation size, as with regular ILS.

We found that GGLS is the best-performing algorithm, and the test results show that its advantage is statistically significant, as shown in Table 6. While SSGLS is also a complex algorithm, it does not offer a statistically significant improvement over ILS. On the other hand, ILS performs comparably to the SSGLS despite being simpler; we believe this is the key takeaway from this assignment.

Appendices

1. Relevant implementation files

This is a copy of README.md file in the solution directory. We provided it here for easy access.

This solution contains the software implementation for the second assignment.
A summary of relevant files and folders is given below. For more details, please see the documentation in the files.

- **graph.py:** Contains the Graph class. Encapsulates the functionality to load the graph from text representation file and graph operations like applying partitioning as solution, moving nodes and calculating gains.
- **node.py:** Node class file. Contains properties of a node, such as id, neighbors, lock status, etc.
- **node_linked.py:** Implements linked list functionality, with previous, next, insert, remove, etc. Extends Node class.
- **fm_impl.py:** Contains the FM class, which implements the Fiduccia–Mattheyses (FM) heuristic. Provides functionalities like initializing and managing buckets, running a FM pass and generating execution statistics.
- **mls.py:** Implements MLS by running FM instances. Exports the results to pickle files and contains functionality to generate metrics and statistics.
- **ils.py:** Implementation of ILS algorithm.
- **ils_adaptive.py:** Extension of ILS with adaptive pursuit algorithm.
- **gls.py:** Implementation of GLS algorithm.

- **experimentation.py:** Contains methods that executes various experiments.
- **test_methods_1.py:** Contains various test cases for implemented functions.
- **utils.py:** Contains helper methods, such as loading datasets from pickle files, statistical significance test and formatting datasets for display.
- **pckl:** This folder contains the archived experiment results. Archive format is pickle.
- **a_runner_*.ipynb:** Python notebooks for running experiments.
- **a_*.ipynb:** Python notebooks for result analysis, table and chart generations.

2. Unit tests and results

Test Method	Description	Status
test_load_graph	Ensures that the vertices and edges are loaded correctly	Passed
test_cutsize	Tests the cut size determination algorithm. Uses a manually crafted graph.	Passed
test_graph_operations	Tests basic graph operations like moving node and calculating gain.	Passed
test_linked_node	Tests the linked node operations, like setting next and previous nodes, and removing a node from the linked list.	Passed
test_fm_single_pass	Tests the FM algorithm for a single pass. It uses a manually crafted graph, with known best solution. The test will check the bucket distribution.	Passed
test_fm_run	Tests the FM algorithm for a full run. It uses a manually crafted graph, with a known best solution. The algorithm is expected to find the best solution.	Passed
test_edge_case_start_optimal	Tests the FM algorithm with a starting solution that is already optimal. Expected is the algorithm does not change the solution.	Passed
test_graph_with_island	Tests the FM algorithm with a not fully connected graph. It uses a manually crafted graph, with a known best solution. The algorithm is expected to find the best solution.	Passed
test_fm_run_500	Tests the FM algorithm for a full run. It uses the target graph, Graph500.txt. We check the initial cut size and the final cut size. The solution must be balanced as well.	Passed
test_significance	This method tests the implementation of Mann-Whitney U test. For testing we calculate one sample from recorded data manually and test against API.	Passed

Table 7: Unit tests and results

3. LLM prompt samples

File: ils.py, method: analyze_ils_performance

```
#LLM Prompt: Iterate over the given files in the folder and load the results from
each file. Use utils.load_ils_results_from_pickle method.
```

```
files = os.listdir(pickle_folder)
    if additional_files:
        files.extend(additional_files)
    for file in sorted(files):
        if file.endswith('.pkl'):
            file_path = os.path.join(pickle_folder, file)
            results = utils.load_ils_results_from_pickle(file_path)[1]
```

File: mls.py, Method: run_mls_parallel

```
# LLM Prompt: Introduce a new run_mls_parallel function that runs the for-loop in
run_mls function in parallel.
```

```
def run_mls_parallel(max_iterations=10000, runs:int=10, graph_file="Graph500.txt"):
        global max_iter
        global graph_filename
        graph_filename = graph_file
        max_iter = max_iterations

        with Pool() as pool:
            results_list = pool.map(single_run, range(runs))

        best_cuts, results = zip(*results_list)
        best = min(best_cuts)

        return _process_results(results, best, max_iterations, runs,"MLS-parallel")
```

File: node_linked.py, Class: LinkedNode

```
#LLM prompt: to generate the class skeleton:generate a LinkedNode class. inherit
from Node class. It needs next and previous properties.
#We adjusted the rest manually.
```

```
class LinkedNode(Node):
    """A node class that inherits from Node and adds linked list functionality"""

    def __init__(self, node_id: int, x_in: int = 0, y_in: int = 0):
        super().__init__(node_id, x_in, y_in)
        self.next = None
        self.prev = None
```

Practical_Assignment2.ipynb, generating plot for MLS results

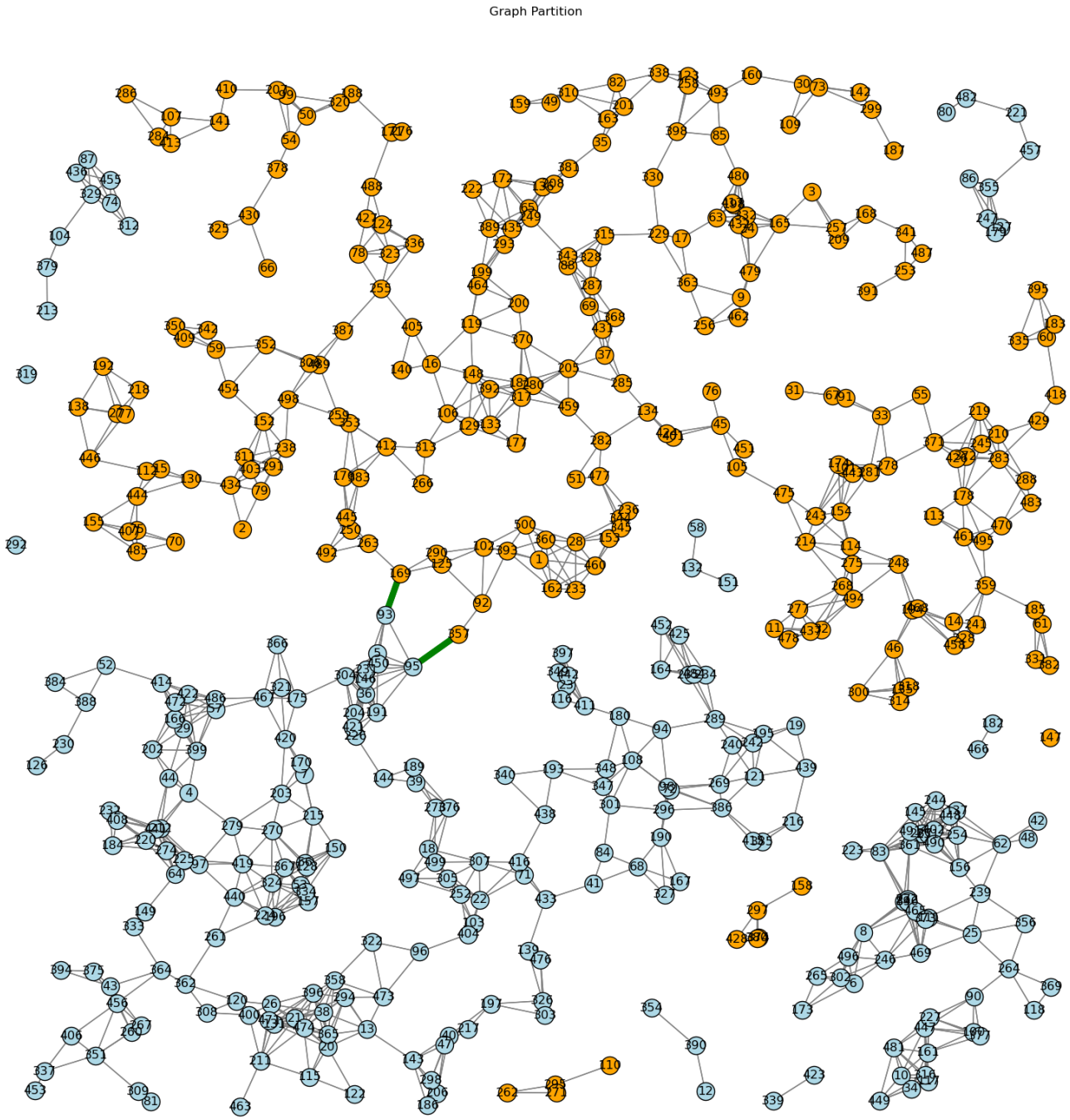
```
#LLM Prompt: Given the 3 code lines below:
# cut_sizes_mls = list(mls['Best Cut'])
```

```
# stdev_mls = np.std(cut_sizes_mls)
# mean_mls = np.mean(cut_sizes_mls)
# generate a line chart. x - axis is the cut size, y axis is the item index. Labels = ['Cut Size', 'Run Number']. Include stdev and mean as vertical dashed lines. color of mean line is blue, stdev is orange.
```

```
# Create x-axis values (run numbers/indices)
run_numbers = list(range(1, len(cut_sizes_mls) + 1))

# Create the figure and axis
plt.figure(figsize=(10, 6))

# Plot the line chart
plt.plot(run_numbers, cut_sizes_mls, marker='o', linestyle='-')

# Add horizontal lines for mean and standard deviation
plt.axhline(y=mean_mls, color='blue', linestyle='--',
            label=f'Mean: {mean_mls:.2f}')
plt.axhline(y=mean_mls + stdev_mls, color='orange', linestyle='--',
            label=f'Mean ± StdDev: {stdev_mls:.2f}')
plt.axhline(y=mean_mls - stdev_mls, color='orange', linestyle='--')

# Add labels and title
plt.xlabel('Run Number')
plt.ylabel('Cut Size')
plt.title('Cut Size per Run with Mean and Standard Deviation')
plt.grid(True, linestyle='--', alpha=0.7)
plt.legend()

# Display the plot
plt.tight_layout()
```

4. Best-Found solution: Graph

The best-found solution with cut size equal to two visualized. The required edges to be cut between partitions are highlighted in green.


Graph Partition