Evolutionary Computing

Graph bi-partitioning assignment

Master's in Artificial Intelligence
Graduate School of Natural Sciences
Utrecht University

Odysseas Papakyriakou

7/04/2022

1. Introduction

The purpose of this assignment is to compare four different meta-heuristic algorithms on their performance for solving the graph bi-partitioning problem.

1.1. Graph bi-partitioning

1.1.1. The graph bi-partitioning problem

Given a graph G of vertices and undirected edges connecting two vertices, the goal is to divide the set of vertices in two equally sized subset such that the number of edges that connect two vertices belonging to the two different subsets is minimized.

The planar graph used for comparing the algorithms consists of 500 vertices and is specified in the file Graph500.txt. Each line of the text file gives the ID number of the vertex, its coordinates in the plane, the number of connected vertices, and the ID numbers of the connected vertices. For instance, the line:

1 (0.502987, 0.528829) 8 28 102 162 233 360 393 460 500

states that the vertex 1 is connected to 8 other vertices with indices 28 102 162 233 360 393 460 500. The coordinates are not used in this project.

1.1.2. Solution representation

Solutions to the graph bi-partitioning problem are represented by binary strings $x_1 \ldots x_l$ with l the number of vertices in the graph and $x_i \in \{0, 1\}$ specifying to which of the two partitions the vertex i belongs. Since the partitions need to be of equal size, the number of ones and zeroes in each solution string needs to be equal to $l/2$.

1.2. Heuristic and meta-heuristic algorithms

Local search (aka heuristic) algorithms iteratively change a solution until no better solution is found in the neighborhood of the current solution. Therefore, the resulting solution of a local search algorithm is a local optimum in the neighborhood of the current solution. The local search algorithm used in this project is the Fiduccia-Mattheyses (FM) heuristic, which is known for its effective time complexity in examining different partitions for the graph. Traditionally, it uses a doubly linked list, allowing insertion and deletion in constant time (Fiduccia & Mattheyses, 1982). In this implementation, a hash map data structure was used instead, again allowing insertion and deletion in constant time. More details on the implementation of the FM algorithm are provided in the following sections.

The three meta-heuristic algorithms aim to improve the performance of the FM algorithm. These algorithms are introduced very briefly here, and they are further explained in the following sections.

1. *Multi-start Local Search* (MLS) simply restarts local search from a set of randomly generated initial solutions. The best local optimum is returned as final solution.
2. *Iterated Local Search* (ILS) mutates the current solution found by local search and applies local search again from the mutated solution. When the new local optimum is better than the

current one, ILS continues its search from this new local optimum, else it simply returns to the previous local optimum. The size of the mutation in this project was determined experimentally.

3. *Genetic Local Search* (GLS) applies selection and recombination to a population of local optimal solutions obtained by a local search algorithm, in this case the FM heuristic. After applying recombination, the offspring is optimized with the FM heuristic to become a new local optimum. Thus, the solutions in the population of GLS are all local optimal solutions. Selection focuses the search towards good regions in the search space, while recombining local optima generates good starting points for the local search algorithm (the FM) in order to find new local optima.

A variation of the MLS algorithm was also introduced, which will be called Depth-First MLS.

4. *Depth First MLS* (DF-MLS) is similar to MLS in that it explores a larger variety of solutions. However, it aborts solutions if no reasonable improvement is found within a given time-frame.

2. Experimental study

To investigate the performance of the four algorithms, two experiments were conducted. For each experiment, each algorithm tried to solve the graph bi-partitioning problem 25 times and the mean results were compared. The statistical significance test that was used to check whether the observed differences were statistically significant was the Mann-Whitney U test, the non-parametric version of the independent samples t-test. The significance level α was conventionally set to 0.05.

The processor on which the experiments were run was a Ryzen 7 5700 series.

2.1. Experiment 1

For the first experiment, the algorithms were compared on the basis of generating the same number of FM passes, namely 10.000 passes. This means that the total number of allowed FM passes was fixed to 10.000, so for example, the MLS algorithm could examine 120 different solutions with 10.000/120=83 FM passes each.

2.2. Experiment 2

For the second experiment, the algorithms were compared on the basis of the same run time (physical clock time). This time was determined by the time it took the MLS algorithm to run once, with 10.000 FM passes. The ILS, the GLS, and the DF-MLS algorithms were then allowed to run for this same amount of time. If the algorithms had surpassed the allowed time, they were let to finish the current FM pass before terminating.

2.3. Hypotheses

2.3.1. MLS

MLS is the most simple approach of the four. It is expected that the mean minimum amount of cuts will be higher than that of ILS, GLS, and DF-MLS, because the algorithm does not learn about the structure of the fitness landscape.

2.3.2. ILS

ILS attempts to learn about the fitness landscape by applying increasingly large perturbation sizes to a generated local maximum such that it moves the solution just outside of the region of attraction of that optimum. It is expected that this search will be most successful when the perturbation size is increased every time a solution returns to its previous optimum because this indicates that the perturbation was not large enough to make it leave the field of attraction. As such, it is expected that ILS will perform slightly better than MLS.

2.3.3. GLS

GLS is the most computationally expensive algorithm because it does not only use the FM heuristic, but it also applies selection and recombination. However, because of its ability to learn the fitness landscape quite well, it is expected that it will be the best performing algorithm, outperforming MLS, ILS, and DF-MLS.

2.3.4. DF-MLS

Due to its similarity with MLS, DF-MLS is expected to outperform MLS, but not more complex algorithms, namely the ILS and the GLS.

3. Description of the program

The archive contains two main directories:

1. results
2. src

3.1. results

The results directory contains 10 files, five for each experiment. The boxplots is a visualization of how the four algorithms performed in the 25 runs; the data file shows the minimum cut found in each of the 25 runs; the times file shows the cpu time it took each algorithm to perform the 25 runs; the descriptives file shows the mean, median, standard deviation and cpu time of each algorithm over the 25 runs; and the stat_results file shows all six unique pairs of algorithms that were tested with the Mann-Whitney U test, along with the U statistic and the p-value.

3.2. src

The src directory contains the source code. This includes the text file that represents the graph problem, the creation of all the algorithms, the statistical tests, and the experiments. Specifically:

3.2.1. Fiduccia_Mattheyeses_algorithm

The Fidiccia_Mattheyeses_algorithm directory contains two Python files: the FM.py, containing all the methods used to run the FM algorithm, and the runFM.py, which performs as many passes of the FM as given by input. The runFM.py also accepts two other paremeters: solution, which is the solution to be optimized with the FM, and threshold, which is the number of unimproved passes the algorithm is allowed to run before it stops.

3.2.2. meta_heuristic_algorithms

The meta_heuristic_algorithms directory contains one Python file for each meta-heuristic algorithm, namely Multi-start Local Search (MLS.py), Iterated Local Search (ILS.py), and Genetic Local Search (GLS.py). Besides these three algorithms, an improved version of the MLS was also compared, which was called Depth-First MLS, and its code is within the MLS.py file.

Both of these directories also contain an __init__.py file that initializes the directories as a module. This helps to import the methods in different files.

3.2.3. experiments.py

This file brings together all previously described meta-heuristic algorithms to run the experiments. The number of runs is specified when instantiating an experiments object, and in this case it was set to 25.

3.2.4. helpers.py

This file contains a method to read and process the graph data.

3.2.5 vertex_class.py

This file is used for the representation of a vertex. It gives each vertex five attributes: number, which is the unique integer number of that vertex and ranges between 1 and 500; partition, which is the partition to which the vertex belongs, and can be either 0 or 1; gain, which is the gain in the number of cuts when that vertex changes partition; neighbors, which is a list of numbers of vertices between 1 and 500, to which the vertex is connected; cuts, which is how many neighboring vertices are in the other partition compared to the current vertex.

3.2.6. statistical_tests.py

This file generates the descriptive statistics, the boxplots, and runs the Mann-Whitney U test on all six unique pairs of algorithms.

3.2.7. main.py

Here is were the code is run. It runs both experiments, generates the descriptive statistics and the boxplots, as well as running the statistical tests.

4. Algorithms

4.1. Fiduccia-Mattheyses (FM) algorithm

This algorithm is quite complex and can be confusing to readers who are not familiar with it. A very nice visual explanation of the algorithm is provided here: https://www.youtube.com/watch?v=CKdc5Ej2jSE

The FM algorithm is a particularly useful local search algorithm because of its ability to apply insertion and deletion in constant time when examining different partitions of the graph. Traditionally, this was done by using a doubly linked list, but in this implementation the use of a hash map was preferred, again allowing for insertion and deletion in constant time. Since the programming language used was Python3, the hash map was implemented with the native dictionary data structure.

Particularly, three different dictionaries were used. Dictionaries are made up of a key and associated values, allowing search, addition and deletion in constant time, O(1). The dictionaries were used to represent the vertex data and the two buckets. For the vertex data , the key is represented by a string of the number of the vertex because this is easy to hash. As a value, data about the vertex was used, such as as gain, partition and neighbors (the vertices to which one vertex is connected). For the two buckets, the number of a vertex was used as a key because in this way the vertex data dictionary can be linked in O(1) to the bucket dictionaries. Sets were used as the data structure of the value because sets are implemented as a hashmap in Python, again allowing search, insertion and deletion in constant time, O(1). Both buckets were initialized with the same amount of vertices to keep the solution balanced at the start.

For one FM pass the algorithm goes trough both buckets in a balanced way by first taking one from the right bucket and then one from the left. This means that at the even iterations the partition will be even and at the odd iterations the partition will be odd. So, at the end only even numbers are considered as valid solutions (because the partition needs to be of equal size).

An iteration of the implemented FM pass:

1. Remove the element from the bucket
2. Get removed element in the vertex data
3. Change partition of removed element
4. Get neighbors of removed element
5. Search neighbors of removed element in buckets
6. Update the gain of the neighbors based on the changed partition in step 3
7. Change total amount of cuts by removed element gain
8. Update roll back list with removed element

The rollback list was used to undo the vertex changes and roll back to the best cut size encountered throughout the pass

4.2. MLS algorithm

As mentioned previously, the MLS algorithm simply restarts local search from a set of randomly generated solutions. Thus, it is not purposefully driven to a better area of the fitness landscape, since it is constrained by the randomly generated solutions. Therefore, the size of these solutions is important in determining the performance of MLS. Since a total of 10.000 passes is allowed, the number of randomly generated solutions to be improved with the FM heuristic is given by 10.000/passes_per_solution. If the number of passes per solution is too small, then the initial randomly generated solution will not by improved by much. Also, if the number of passes per solution is too large, then there will be a very shallow exploitation of the fitness landscape. Thus, the population size was decided to be 120, giving 83 FM passes per solution. This means that each of the 120 randomly generated initial solutions will be improved by 83 FM passes.

4.3. ILS algorithm

ILS is dependent on the perturbation size that is applied to the local optimum, as found by the FM heuristic, so its selection is very important. The best pertubation size should “kick” the local solution from its local optimum to another local optimum close by. To find this perturbation size 5 initial random solutions were created. Next, different copies from these solutions were created, for an amount of 10 different perturbation sizes: [0.15,0.1,0.09,0.08,0.07,0.05,0.04,0.03,0.02,0.005]. The perturbation size that seemed to improve the solution the most was 0.05, so this was applied in both experiments.

4.4. GLS algorithm

The Genetic Local Search (GLS) meta-heuristic algorithm is applied to the space of local optima. This means that the initial population for the GLS is made up of solutions that are already optimized with the FM algorithm. For these experiments, 50 randomly generated solutions were optimized with 100 passes of the FM algorithm. In each iteration of the GLS, two parents are randomly selected, and if their Hamming distance is greater than half of the string-length (so greater than 250), one of the parents has its bits flipped. The similar bits between the two parents are passed on to the child, and the different ones are randomly generated while maintaining an equal number of 1’s and 0’s. This was done by first generating the appropriate number of 1’s and 0’s, and then randomly placing them in the unfilled places of the child string.

Therefore, the unfilled places of the child-solution are filled with uniform crossover, which is quite disruptive. This usually results in a worse solution than the two parents, which had already been optimized with the FM algorithm. Thus, the child is also optimized with the FM heuristic. If the resulting child does not already exists in the population and if it is better than the worst solution in the population, it replaces it. Across all 25 runs of each experiment, the initial population was kept the same. This was particularly needed for the second experiment, where the allowed cpu time is fixed. If the initial population was created for every run, it would take a considerably larger amount of time to complete the first experiment and it would only run around 10 iterations for each run in the second experiment.

For the first experiment, GLS run for 100 iterations, thus performing 100 FM passes to each child to maintain a total of 10.000 FM passes. For the second experiment with a fixed amount of time, GLS was adapted to perform as many iterations as possible by setting a relatively low threshold for the

amount of FM passes allowed. Particularly, if there had been more than 7 FM passes without an improvement, the algorithm kept the best solution and moved on to the next child. This allowed to run for around 1.000 iterations, thus exploring a much wider area of the fitness landscape. As will be shown in the following sections, this was a critical change for the superior performance of GLS.

4.6. DF-MLS algorithm

This algorithm aims to improve on the traditional Multi-start Local Search algorithm, which simply restarts the FM heuristic from randomly generated solutions for a fixed amount of passed. The idea is to give a solution an adaptive patience level, where in the end, leads to fewer FM passes if there is not much improvement as the FM passes increase.

How it works:

Similarly to the traditional MLS approach, a constant c dictates the maximum amount of FM passes allowed. Then, instead of equally distributing the amount of FM passes over the set of generated solutions, the FM passes are performed up until no substantial improvements are made. When that happens, a new random solution is generated to perform the FM passes on. This process is repeated until the number of allowed FM passes is reached. The idea is that the amount of FM passes each solution gets is no longer equally distributed.

Unique mechanics:

One important element is to determine whether a solution is allowed to use up more FM passes, (the patience of the algorithm). Too much patience will likely result in one population element using up all FM passes, making incredibly small improvements with each pass. On the other hand, too little patience could prematurely cut a solution off. A fixed number patience level was opted for. First, a minimum cut improvement t was set. Then, after each FM pass the cuts of the solution are evaluated. If the difference d in cuts before and after the FM pass is greater than t, the solution is allowed to carry on. If it is smaller, which means it has not improved enough, patience level is decreased. If patience reaches zero, the solution is terminated and a new solution is generated.

A bonus bound and a bonus multiplier were also set. The bonus bound sets a bound (in number of cuts) from which the algorithm will provide bonus patience allocation to a solution. It is important to note here that patience equals a solution's entitlement to FM passes, as described in the patience mechanism above. By setting a bonus, we can stimulate promising solutions to improve further. A linear increase of patience allocations is implemented. Setting the bonus bound to 15 (cuts) and the bonus multiplier to 20, a solution would get 20 bonus patience points for each cut below the bound. For example, a solution with 13 cuts would get $(15 - 13) * 20 = 40$ extra patience points to improve and a solution with 6 cuts would get $(15 - 6) * 20 = 180$ patience points before it gets terminated.

5. Experimental results

This section displays the results of the two experiments. For both experiments, the statistical test that was conducted for all pairs of algorithms was the Mann-Whitney U test, the non-parametric version of the independent samples t-test. The non-parametric alternative was preferred because of the small sample size, n = 25. The null hypothesis is that there is no difference in the performance

of the two algorithms over the 25 runs, while the alternative hypothesis is two sided, namely, that the performance of the two algorithms is different.

For comparison and reproducibility purposes, it is mentioned that the processor on which the experiments were run was a Ryzen 7 5700.

The boxplots depict the mean of the 25 runs for each algorithm with a white circle, while the black points represent the minimum number of cuts found in each run.

5.1. Fixed number of FM passes at 10.000

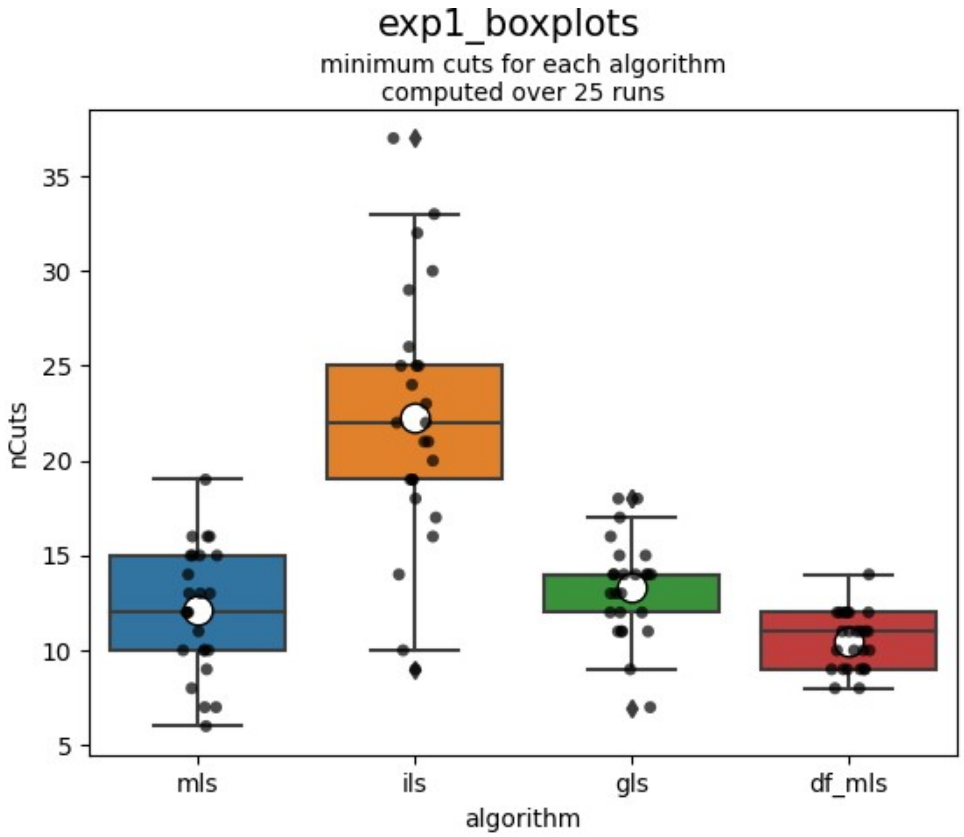
For this experiment, all algorithms were compared based on a fixed number of FM passes, namely, 10.000 passes. Therefore, each of the 25 runs terminated after the algorithm performed 10.000 FM passes.

Table with the descriptive results:

	Min. cut	**Frequency**	**Mean**	**Median**	**std**	**CPU time in sec (25 runs)**	**FM passes**
MLS	6	1	12.16	12	3.34	957.49	10.000
ILS	9	1	22.24	22	6.75	1077.54	10.000
GLS	7	1	13.28	14	2.57	971.7	10.000
DF-MLS	8	2	10.48	11	1.5	1293.17	10.000

Min. cut represents the minimum amount of cuts each algorithm found across all 25 runs, and the frequency represents how many times that minimum was found.

Boxplots taken over 25 runs:



Statistical tests (Mann-Whitney U test):

Algorithms	U statistic	P-value (rounded)
MLS - ILS	53	< 0.001
MLS - GLS	254.5	0.262
MLS - DF_MLS	420.5	0.035
ILS - GLS	562	< 0.001
ILS - DF_MLS	589.5	< 0.001
GLS – DF_MLS	521.5	< 0.001

From the tables it becomes apparent that MLS and GLS are the best performing algorithms. Moreover, hypothesis testing shows that these algorithms are not significantly different from each other at a 0.05 level of significance, $U = 53$, $p < 0.262$, even though MLS found a slightly better solution to the problem. It is important to remember that in this experiment the 10.000 FM passes of the GLS algorithm were equally divided across 100 iterations, resulting in 100 FM passes per iteration, 100*100 = 10.000.

5.2. Fixed run-time experimentally

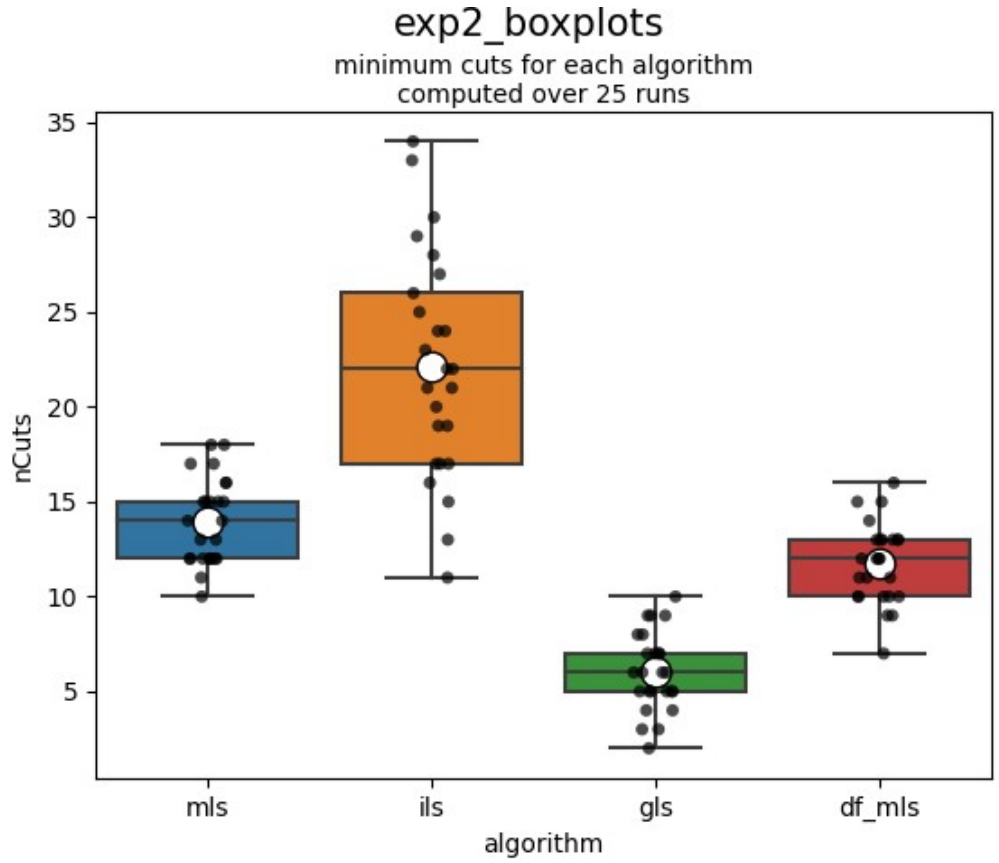
For this experiment, all four algorithms were compared based on the time it took MLS to run 10.000 FM passes. Based on the previous experiment, the CPU time it took MLS to finish its first run with 10.000 FM passes was roughly 41 seconds, so this time was used as a limit for the remaining algorithms. MLS was again let to run for 25 times so there can be a fair comparison between the algorithms. The CPU time between the algorithms fluctuates a tiny amount because the algorithms were allowed to finish their current step once time ran out. Therefore, one run of each algorithm terminated after approximately 41 seconds, and not after any given amount of FM passes, as in the previous experiment.

Table with the descriptive results:

	Min. cut	Frequency	Mean	Median	std	CPU time in sec (25 runs)
MLS	10	1	13.92	14	2.25	957.08
ILS	11	1	22.12	22	5.98	1070.7
GLS	2	1	6.04	6	2.07	956.69
DF-MLS	7	1	11.76	12	2.13	935.84

Min. cut represents the minimum amount of cuts each algorithm found across all 25 runs, and the frequency represents how many times that minimum was found.

Boxplots taken over 25 runs:



Statistical tests (Mann-Whitney U test):

Algorithms	**U statistic**	**P-value (rounded)**
MLS - ILS	60	< 0.001
MLS - GLS	624.5	< 0.001
MLS - DF_MLS	420.5	0.003
ILS - GLS	562	< 0.001
ILS - DF_MLS	589.5	< 0.001
GLS – DF_MLS	521.5	< 0.001

The tables in this experiment clearly show the superior performance of the GLS algorithm, while hypothesis testing shows that GLS is significantly different than every other algorithm. It is important to remember that for this experiment, GLS performed many fewer FM passes per iteration than in the previous experiment, allowing it to run for around 1000 iterations per run.

6. Interpretation of the results

The experiments show how effective the Genetic Local Search algorithm can be in solving the graph bi-partitioning problem. GLS turned out to be by far the best performing algorithm in the second experiment, and as good as MLS in the first experiment. The difference in the performance of the GLS in the two experiments highlights the importance of exploring a wider area of the fitness

landscape as possible. In other words, GLS performed so much better in the second experiment because it run for 1000 iterations compared to 100, thus being able to explore way more local optima. This improvement in the performance of GLS was enables by limiting the amount of FM passes per iteration, such that if there were 7 FM passes where there was no improvement, the algorithm proceeded to the next iteration. Compared to the 100 FM passes across 100 iterations, this change seemed to have been critical for the superior performance of GLS in the second experiment.

7. Conclusion

Conclusively, the best performing algorithm among Multi-start Local Seach, Iterated-Local Search, Genetic Local Search, and Depth-First Multi-start Local Search, was the GLS algorithm. When tuned to explore a wider area of the fitness landscape it found a minimum cut of 2 for the graph bi-partitioning problem, thus by far outperforming all other algorithms.

References

Fiduccia, C. M. & Mattheyses, R. M. (1982). A linear-time heuristic for improving network partitions. *19th Design Automation Conference*.