Parameterless Metaheuristics for the MDMKP

June 11, 2019

1 Goals

Metaheuristics commonly have a wide array of parameters that must be tuned in order to obtain good performace. This task is often time consuming, and can lead to brittle systems ill-equiped to handle changing problems. Instead, we focus solely on metaheuristics that do not require any parameter tuning. These methods often have the added effect of being simple and easy to implement, while still giving competitive results.

2 Algorithms Used

2.1 Jaya

Jaya is a metaheuristic proposed by Dr. Rao of the Sardar Vallabhbhai National Institute of Technology. It was originally developed to work on continous problems, but has been modified to work on binary problems.

Jaya creates a new solution by moving an existing solution towards the best solution found, while simultaneously moving it away from the worst solution found. In the continous space, each movement vector is scaled by some continous value between zero and one. However, as the MDMKP is a binary formulation, there are no fractional movements. Instead, each bit is scaled either 0% or 100% of it's distance vector. Each percentage has an equal probability of happening. More formally, jaya is described as follows:

Given a population X, define *best* as the best performing solution, and *worst* as the worst performing solution. r is the random function, and uniformly selects an element from the set. For every solution S in the population, construct a modified solution according to the following transformation applied to every bit i in S:

$$S'_i = S_i + r(\{0,1\}) * (best_i - S_i) - r(\{0,1\}) * (worst_i - S_i) > 0 \qquad (1)$$

If the new solution is valid and performs better than the old solution, replace it in the population. One iteration of Jaya will attempt this transformation for every solution in the population.

2.2 TLBO

Teaching-Learning Based Optimization is another metaheuristic developed by Dr. Rao for continous problems . It is a two phase algorithm: the Teaching phase, and the Learning phase.

2.2.1 TBO

The first phase, Teaching Based Optimization, constructs new solutions by moving a given solution along the distance vector found by moving from the mean solution to the best solution. The vector is sometimes doubled. Once again, partial movements are impossible, so a 0% or 100% strategy is applied to each individual bit. Formally, it is defined as follows: For every bit index i in a solution S, update a new solution according to:

$$S'_i = \Big(S_i + r(\{0,1\}) * \big(best_i - r(\{1,2\}) * average_i\big)\Big) > 0 \qquad (2)$$

where *best* is the best scoring solution in the population, and *average* is the average of all solutions in the population.

There are two different methods for calculating the average value. The first is to order the population by objective function values, and then use the middle solution as the median average. However, this has a different intent than the continous method: instead of acting as the centerpoint of the population, it is an effectively randomly chosen midtier solution.

Instead, the mean is calculated as it would be in the continous space. This will results in a vector of numbers between 0 and 1. This can be converted to a discrete representation by treating the values as a likelihood chance of being turned on, like so:

$$S'_i = \Big(S_i + r(\{0,1\}) * \big(best_i - r(\{1,2\}) * (r([0,1]) < average_i)\big)\Big) > 0 \qquad (3)$$

Notice that the random function r was given a continous range from 0 to 1, not a discrete set. This probability method has been shown to give better results than the median method in testing.

The TBO transform is used in the same way as the jaya transform: apply to every solution in the population, and replace the solutions if the new solutions have better objective function scores.

2.2.2 LBO

The second phase of TLBO, Learning Based Optimization, acts by selecting two solutions, and moving the worse performing solution some distance along the vector between them [1]. If the new solution has a better score than the worse performing solution, it is replaced. Solutions cannot be the same. In this implementation, a single instance of LBO is guaranteed to act on every solution at least once, like so:

Algorithm 1 Learning Based Optimization

```
for $sol_1 \in X$ do
  $sol_2 = rand(X)$
  while $sol_1 = sol_2$ do
    $sol_2 = rand(X)$
  end while
  $student, teacher = sort(sol_1, sol_2)$
  $new\_student = copy(student)$
  for $i \in 0..length(student)$ do
    $new\_student_i \mathrel{+}= rand(\{0,1\}) * (teacher_i - new\_student_i)$
  end for
  if $score(new\_student) > score(student)$ then
    $X_{student} = new\_student$
  end if
end for
```

In Rao's implementation, a check to make sure the two selected solutions have different objective values is included. This check is omitted from this implementation, as it is arbitrary, and the uniqueness check is sufficient.

LBO can be thought of as a genetic algorithm with two parents. This effect will be later demonstrated in the section 5: Comparing Metaheuristics.

While TBO and LBO are typically used together to form the TLBO algorithm, each can be used individually.

[1] It is worth noting that the vector from the better to the worse solution contains all the same points as the vector from the worse to the better—the only difference is what solution is replaced, if the new solution yields an improved score.

2.3 Genetic Algorithm

The classic Genetic Algorithm for binary problems is to take two solutions, randomly select a breakpoint, and then construct two new solutions like so:

$$\begin{aligned} new\_solution\_1 &= sol\_1[: breakpoint] + sol\_2[breakpoint :] \\ new\_solution\_2 &= sol\_2[: breakpoint] + sol\_1[breakpoint :] \end{aligned} \tag{4}$$

This allows for very fast construction of new solutions, but is limited in the amount of novel solutions it can produce. For this implementation of a genetic algorithm, a probability based approach is used.

The procedure for generating a single new solution is as follows. First, n solutions are selected. Let M be the column-wise mean of the matrix of the selected solutions. The percantage chance of a bit b being turned on in the new solution is given by M_b. More than one solution could be generated from a single probability matrix M, but this implementation does not do so.

The Genetic Algorithm is used to generate new solutions, and if the new solution scores better than its worst performing parent, the parent is replaced. This process continues until some termination criteria is met.

This Genetic Algorithm is not a parameterless method, as the amount of parents must be determined by the user. However, testing on the MDMKP with 2, 3, 4, and 5 parents did show significant and consistent differences between the performance of different numbers of parents.

2.4 Variable Neighborhood Descent

Neighborhood Descent is a local method that acts on only one solution at a time. Neighborhood search works as follows: for a solution S, a neighborhood N of related solutions is generated. The best performing solution from this neighborhood is selected, and if it outperforms S, S is replaced. If S is not outperformed, stop the search. Variable Neighboorhood Descent uses more than one neighborhood.

For this implementation, two neighborhoods are used. The first is based on the flip transformation: for every bit in the solution, generate a new solution with the bit flipped. The second is based on the swap transformation: for every two unalike bits in the solution, generate a new solution with the bits flipped.

2.5 The Repair Operator

MDMKP problems often have very sparse regions of feasibility within the possible solution space. The repair operator is a function that attempts to move an infeasible solution into the realm of feasibility. This method works as so:

Algorithm 2 Repair Operator

```
prev_infeas = $\infty$
curr_infeas = $\infty - 1$
while cur_infeas < prev_infeas do
  feasible_solutions = $\emptyset$
  least_infeas_b = -1
  for b $\in$ solution do
    new_sol = copy(solution)
    new_sol$_b$ = !new_sol$_b$
    if feasibility(new_sol) < curr_infeas then
      curr_infeas = feasibility(new_sol)
      least_infeas_b = b
      if feasible(new_sol) then
        feasible_solutions += new_sol
      end if
    end if
  end for
  if feasible_solutions then
    return best_scoring(feasible_solutions)
  end if
  solution$_least\_infeas\_b$ = !solution$_least\_infeas\_b$
end while
return NULL
```

This a very simple version of the Repair Operator, that has many duplicate operations. In order to be performant in code, it is necessary to cache the infeasibility throughout the algorithm, and update it to account for whatever the currently flipped bit is.

3 Generating an Initial Population

Since the majority of randomly generated solutions for the MDMKP will be infeasible, it can be difficult to generate a viable initial population. The steps to generate a feasible solution are as follows:

1. Create a random permutation I of the numbers $1..n$, where n is the amount of bits in a solution
2. Create a new solution S with every bit disabled
3. For every index i in I, if enabling S_i will not violate a dimension constraint, enable S_i
4. Once every i in I has been checked, run the repair operator on the generated solution S
5. If S is feasible, add it to the set of feasible solutions
6. Create a new solution S with every bit enabled
7. For every i in I, if disabling S_i will not violate a demand constraint, disable S_i
8. Once every i in I has been checked, run the repair operator on the generated solution S
9. If S is feasible, add it to the set of feasible solutions

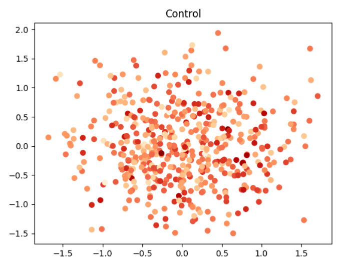
This procedure is run until the target population size has been reached, or some other termination criteria stops the process.

4 Managing Population Diversity

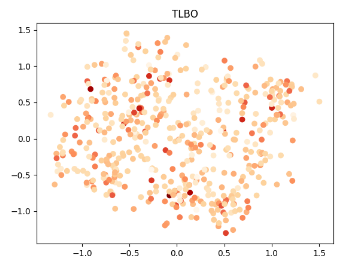
Swarm optimization algorithms rely on creating new solution based off of existing solutions. As such, it is important that the diversity of a population of solutions is preserved. As they are currently implemented, swarm optimization algorithms will converge to several local minima. For example, compare the Principle Component Analysis of the initial population, and the population produced by TLBO.

The clustering effect is less visible on a higher dataset with more variables. To determine if this is due to the clustering being less prevelant on problems with more variables, or PCA being less effective, it is necessary to create a metric to measure diversity.

The diversity metric is defined at Algorithm 3.

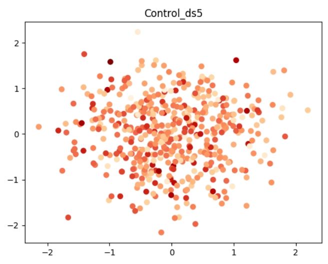


(a) Initial Population

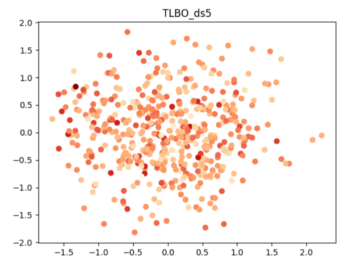


(b) TLBO Population

Figure 1: Example of Metaheuristic Clustering Behavior



(a) Initial Population



(b) TLBO Population

Figure 2: Example of Metaheuristic Clustering Behavior on a Higher Dataset

Algorithm 3 Diversity Metric

```
total = 0
for solution ∈ population do
    distances = [euclidean_distance(solution, sol2) for sol2 in population]
    sort(distances)
    total += sum(distances[1:4])
end for
return  total÷(length(population)*3)
```

The average diversity metrics for the results for the Beasley MDMKP dataset are as follows:

Table 1: Metaheuristic Diversity Results

complete diversities	dataset 1	dataset 2	dataset 3	dataset 4	dataset 5	dataset 6	dataset 7	dataset 8	dataset 9	average
LBO	2.12	2.33	2.47	2.15	2.29	2.41	1.31	2.21	2.32	2.18
jaya	2.04	2.26	2.39	2.09	2.28	2.41	1.45	2.29	2.47	2.19
GA2	2.19	2.34	2.42	2.21	2.31	2.38	1.31	2.27	2.3	2.19
TLBO	2.13	2.34	2.52	2.16	2.37	2.45	1.36	2.3	3.38	2.33
TBO	3.18	4.85	6.99	3.35	5.25	7.53	2.14	5.45	8.77	5.28
control	4.85	7.45	10.1	4.94	7.6	10.47	2.78	8.0	11.02	7.47

Here, the "control" result is VND, and its population is used as the initial population for the other metaheuristics, for reasons that will be later discussed. We therefore see that all the metaheuristics tested are intensification techniques. As population-based metaheuristics rely on creating new solutions from diverse inspiration solutions, a loss of diversity will lead to a loss of effectiveness. It is therefore beneficial to devise a method to limit the progressive diversity loss.

There are several methods to limit loss of diversity, such as expanding the solution pool, reintroducing old, poorly performing outlier solutions to the population, or ensuring that any new solution replacing an old solution must be suffiently different to every other solution in the population, which is the strategy used in this implementation.

To ensure that new solutions are significantly unique, the VND local search is attempted on every solution generated by other metaheuristics. This makes it impossible for two solutions differing in only one bit to be a part of the population, and has the added benefit of reducing the search space from finding the best solution, to finding the best solution watershed. This increases diversity of the resulting populations for every metaheuristic except TBO:

Table 2: Metaheuristic Local Search Diversity Results

complete diversities	dataset 1	dataset 2	dataset 3	dataset 4	dataset 5	dataset 6	dataset 7	dataset 8	dataset 9	average
LBO	2.12	2.33	2.47	2.15	2.29	2.41	1.31	2.21	2.32	2.18
jaya	2.04	2.26	2.39	2.09	2.28	2.41	1.45	2.29	2.47	2.19
GA2	2.19	2.34	2.42	2.21	2.31	2.38	1.31	2.27	2.3	2.19
TLBO	2.13	2.34	2.52	2.16	2.37	2.45	1.36	2.3	3.38	2.33
jaya_ls	2.46	2.93	3.94	2.6	3.21	4.84	1.74	3.78	5.7	3.46
GA2_ls	2.53	2.85	4.02	2.63	3.15	5.0	1.6	3.43	6.02	3.47
LBO_ls	2.49	2.88	4.03	2.61	3.16	5.02	1.65	3.61	6.02	3.5
TLBO_ls	2.51	2.97	4.46	2.63	3.35	5.38	1.64	4.1	6.53	3.73
TBO_ls	2.66	3.75	5.59	2.9	4.1	6.12	2.02	4.87	6.98	4.33
TBO	3.18	4.85	6.99	3.35	5.25	7.53	2.14	5.45	8.77	5.28
control	4.85	7.45	10.1	4.94	7.6	10.47	2.78	8.0	11.02	7.47

Combining a local search with other metaheuristics is a strategy that has been tried before, for example by Dr. Vasko et. al. [1]. However, Dr.

Vasko et. al. created a hybrid method consisting on a local search followed by some other metaheuristic. Testing has shown that this implementations method of embedding a local search, rather than prefixing one, reduces the percent deviation from the optimal by five to ten times.

The local search is also embedded into initial population generation, which is why "control" is VND, and is used as the source population for other metaheuristics.

5 Comparing Metaheuristics

6 Creating a Hybrid Method

A hybrid method consisting of many metaheurisitcs chained together has the potential to outperform any one metaheuristic, but the order of metaheuristics must be chosen with care. Putting a strongly intensifying metaheuristic too early in the chain can cripple the population for the next metaheuristic. This can be somewhat mitigated by running each metaheuristic only once before looping through the chain, as opposed to running one technique to exhaustion before moving on to the next. However, the order can still have a significant effect.

Using the table of diversities, as well as the general principle that intensification and diversification should oscillate, and the goal of including each metaheuristic only once, the patterns TLJB, TBJL, JBTL, and TLJB result. Keeping TBO and LBO consecutive due to the effectiveness of the TLBO method, and including the GA twice, once with two parents, once with four parents, gives the chain GA[parents=2], Jaya, GA[parents=4], TBO, LBO.

7 Runtime Parameters

Despite using primarily parameterless algorithms, choices still must be made while computing the results.

Maximum Failed Attempts When testing a metaheuristic, a single run through is often not enough to extract the full benefits of the method. Instead, it is ran repeatedly, until it fails to produce an improvement for any solution in the swarm $max\_failed\_attempts$ times in a row. This setting will directly impact the run time of the algorithm. In this implementation, a high value of 25 is used. This can result in algorithms taking up to three hours, but there is an additional time constraint.

Time Constraint Each algorithm is tested for 10 seconds, and for 60 seconds. Due to the way the time is measured, the true runtime can be up to three seconds over the limit. The Maximum Failed Attempts stop parameter can also cause the runtime to be much less than the allotted time.

Use of Helper Function The helper function are the Repair Operator, and the embedded local search. Testing has shown that the Repair Operator always improves results, but the local search can occasionally negatively impact the results. For this reason, the Repair Operator is always enabled, and two versions of each algorithm are tested—one with local search, and one without.

Genetic Algorithm Parents Testing of different amounts of parents did not show a large impact on the performance, so here, only two parents are used, except in the hybrid method, which has one two parent instance and one four parent instance.

8 Results

Below are tables showing the percent deviation from optimal for each algorithm. Each algorithm had the same initial swarm, generated using an embedded local search. "Control" is therefore equivalent to the VND metaheuristic.

Optimal values were calculated using a CPLEX program given two hours to search for the best solution. For datasets 5, 7, 8, and 9, the metaheuristics found solutions better than the CPLEX reported optimals. These negative percentages result in a misleadingly low optimal.

Additionally, dataset 7 has many problems that are extremely difficult to find feasible solutions for. Therefore, 7 is more of a test of the initial population generation algorithm than any specific metaheuristic. If the initial population generator finds a solution CPLEX did not, it is reported as 0% error, but if the inverse happens, it is 100% error. This leads to a deceptively high average.

Averaging only datasets 1, 2, 3, 4 and 6 from the hybrid method with embedded local search gives an overall lowest percentage of 0.27%.

Table 3: Metaheuristic 10 Second Results

complete percentages	dataset 1	dataset 2	dataset 3	dataset 4	dataset 5	dataset 6	dataset 7	dataset 8	dataset 9	average
GA2_ls	0.09	0.43	1.47	0.3	0.84	2.24	10.78	2.56	3.47	2.46
TLBO_ls	0.13	0.56	1.25	0.44	0.93	1.87	11.26	2.54	3.37	2.48
jaya_ls	0.3	0.49	1.39	0.77	0.83	2.22	11.79	2.42	3.33	2.62
LBO_ls	0.17	0.46	1.65	0.4	0.87	2.5	11.21	2.6	3.94	2.64
GA2	0.29	0.33	0.43	0.64	0.67	0.88	11.08	3.83	6.18	2.7
TBO_ls	0.33	0.89	1.7	0.81	1.11	2.15	12.29	2.9	3.5	2.85
LBO	0.48	0.53	0.53	0.89	0.79	0.98	11.7	3.82	6.31	2.89
TLBO	0.53	0.54	0.96	0.9	0.95	2.01	11.4	6.74	6.73	3.42
jaya	1.5	1.3	1.02	2.26	1.89	1.66	12.73	4.72	5.75	3.65
TBO	2.19	2.94	4.01	2.84	3.3	4.38	13.35	8.3	6.94	5.36
control	6.5	6.67	6.33	6.72	5.76	5.78	15.64	10.14	7.36	7.88

Table 4: Metaheuristic 60 Second Results

complete percentages	dataset 1	dataset 2	dataset 3	dataset 4	dataset 5	dataset 6	dataset 7	dataset 8	dataset 9	average
GA2_ls	0.09	0.17	0.32	0.23	0.35	0.8	10.45	0.31	1.32	1.56
GA2	0.33	0.32	0.29	0.61	0.55	0.47	11.03	0.47	0.66	1.64
LBO_ls	0.16	0.2	0.34	0.4	0.38	0.79	11.15	0.51	1.31	1.69
TLBO_ls	0.12	0.23	0.46	0.42	0.44	0.8	11.02	0.66	1.32	1.72
jaya_ls	0.35	0.33	0.38	0.69	0.54	0.82	11.6	0.85	1.18	1.86
LBO	0.51	0.55	0.49	0.93	0.82	0.73	11.71	0.81	0.72	1.92
TLBO	0.51	0.55	0.63	0.9	0.82	0.75	11.56	1.02	2.08	2.09
TBO_ls	0.37	0.53	0.97	0.77	0.7	1.18	12.06	1.42	1.68	2.19
jaya	1.34	1.16	1.05	2.16	1.84	1.51	12.63	3.49	2.16	3.04
TBO	2.27	2.95	3.66	2.96	3.2	3.8	13.23	5.97	6.17	4.91
control	6.54	6.56	6.37	6.9	5.95	5.73	15.59	10.02	7.38	7.89

Table 5: Hybrid 60 Second Results

complete percentages	dataset 1	dataset 2	dataset 3	dataset 4	dataset 5	dataset 6	dataset 7	dataset 8	dataset 9	average
G2JG4TL_ls	0.06	0.15	0.3	0.14	0.33	0.7	11.22	0.29	1.14	1.59
G2JG4TL	0.17	0.24	0.26	0.48	0.46	0.46	11.61	0.4	0.63	1.64
control	6.25	6.52	6.31	7.1	6.03	5.9	16.2	9.81	7.11	7.91

8.1 Additional Visualizations

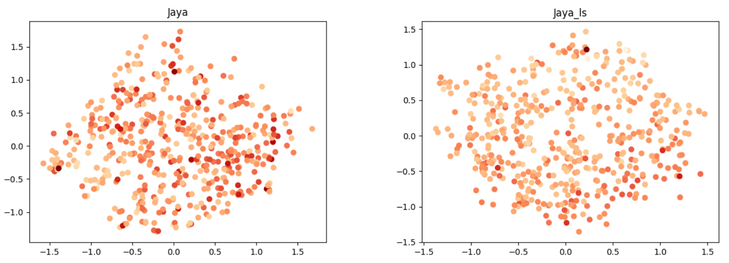


Figure 3: Jaya and Jaya Local Search

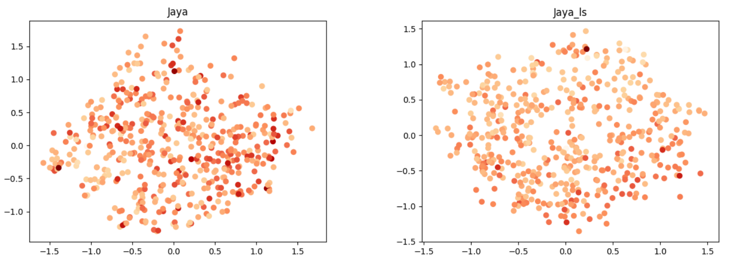


Figure 4: Jaya and Jaya Local Search

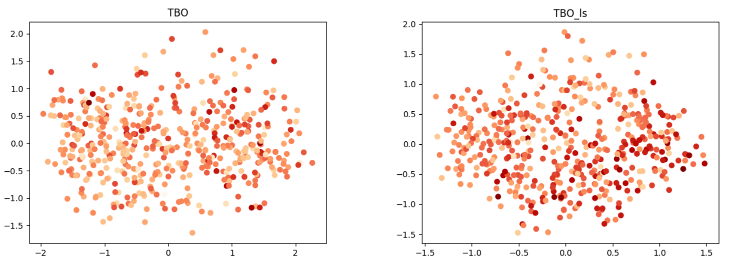


Figure 5: TBO and TBO Local Search

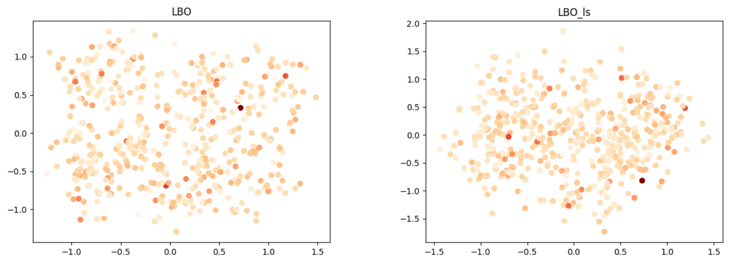


Figure 6: LBO and LBO Local Search

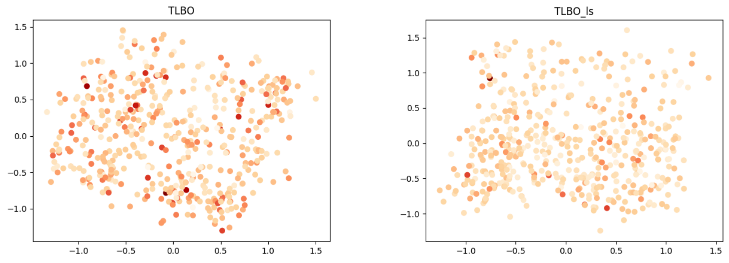


Figure 7: TLBO and TLBO Local Search

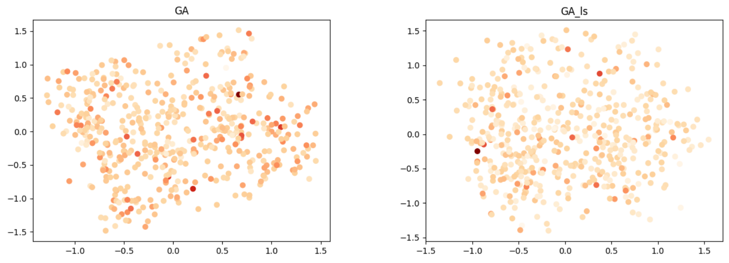


Figure 8: GA and GA Local Search

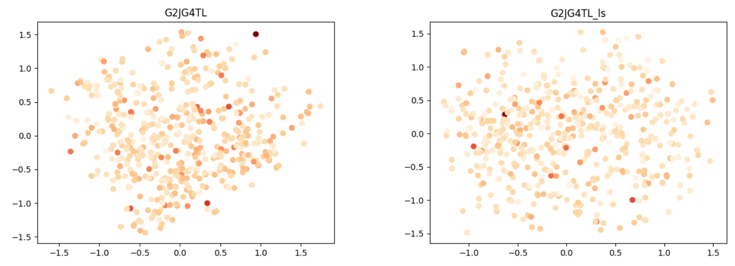


Figure 9: G2JG4TL and G2JG4TL Local Search

References

[1] Francis J. Vasko, Yun Lu, and Kenneth Zyma. An empirical study of population-based metaheuristics for the multiple-choice multidimensional knapsack problem. *Int. J. Metaheuristics*, 5(3/4):193–225, January 2016.