1. **3 Main changes has been implemented in the code.**
2. Reproduction Phase – Crossover

Sols matrix is used in unsorted shape however in genetic reproduction it is necessary to identify strong members. With using sortindex, our populations reproduction is far more efficient now.

By this: child = reproduction(sols[p1,:].tolist(),sols[p2,:].tolist(),l,u)

To this: child = reproduction(sols[sortindex[p1],:].tolist(),sols[sortindex[p2],:].tolist(),l,u)

1. Cleaning unnecessary-useless members

In previous version, delete operation was performed first and append operation of new reproduced members were later. That was causing the loss of valuable members. The solution was appending new members, making a new calculation and then deleting useless members.

1. Mutation Phase

In mutation phase, original members were replaced by newly produced members without knowing the new ones are better.That was causing loss of valuable members. The solution was appending new mutated members, making a new calculation and then deleting useless members.

From this: sols[sortindex[ind],:] = mutation(sols[sortindex[ind],:],prob,np.int(size\*ratio))

To this: sols = np.row\_stack((sols, mutation(sols[sortindex[ind],:],prob,np.int(size\*ratio))))

**Performance**

I have tried 10 dimension, 3 constraints 5 problems and I coded an optimal solver to compare performances.

**Problem 1**

Optimal Solution is:

[0, 0, 1, 1, 1, 0, 0, 0, 1, 1]

14

Original GIPSolver

[1. 1. 0. 0. 1. 1. 0. 1. 0. 0.]

7.0

Modified GIPSolver

[1. 1. 0. 0. 1. 1. 0. 1. 0. 0.]

7.0

**Problem 2**

Optimal Solution is:

[1, 1, 0, 0, 0, 1, 0, 1, 1, 1]

19

Original GIPSolver

[0. 1. 1. 0. 1. 0. 1. 0. 0. 1.]

11.0

Modified GIPSolver

[0. 1. 1. 1. 1. 0. 1. 0. 0. 1.]

11.0

**Problem 3**

Optimal Solution is:

[0, 0, 1, 0, 0, 1, 1, 1, 0, 1]

17

Original GIPSolver

[1. 1. 0. 1. 0. 0. 0. 0. 1. 1.]

5.0

Modified GIPSolver

[1. 1. 0. 1. 0. 0. 0. 0. 1. 1.]

5.0

**Problem 4**

Optimal Solution is:

[0, 0, 0, 1, 1, 0, 1, 0, 0, 1]

12

Original GIPSolver

[1. 0. 0. 1. 0. 1. 0. 1. 0. 0.]

5.0

Modified GIPSolver

[1. 0. 0. 1. 0. 1. 0. 1. 0. 0.]

5.0

**Problem 5**

Optimal Solution is:

[0, 0, 1, 1, 0, 0, 1, 0, 1, 1]

19

Original GIPSolver

[1. 0. 1. 1. 1. 1. 0. 1. 0. 0.]

13.0

Modified GIPSolver

[1. 0. 1. 1. 1. 1. 0. 1. 0. 0.]

13.0

So; unfortunately there is no significant performance improvement in terms of closeness to optimal solution.Moreover, the changes that we made are increased run time. However, it is probabilistic solver; it could be case specific for this type of problems and possible to produce more relevant solutions in other dimensions or constraints.

b) **cx = d constraint**

I tried to manipulate scoring algorithm an add a modified penalty function. It was turning higher values to 0 which gives the toughest penalty to positive values. But with an absolute value penalty function which is not sensitive to the sign of calculated score, both sides take place in maximization and not always but generally equality constraints have been assured. New constraint is basicly like this:

for i in range(size+2):

LHS[:,i] = LHS[:,i] - np.transpose(b)

LHS = LHS\*100

score = np.dot(c,np.transpose(sols)) + np.absolute(LHS.sum(axis=0))

With high number of constraints and high number of dimensions, an equation set with optimal solution is hard to find. So, I decided to use it with 1 equality constraint in dimension 5. (As recommended)

Here is my solution and optimal solution comparison:

Problem -1

C = array([2, 2, 1, 1, 1])

A = array([[3, 3, 4, 0, 2]])

b = [8]

Optimal Solution is:

[1, 1, 0, 1, 1]

6

Modified Equality GIPSolver

[1. 1. 0. 0. 1.]

5.0

Problem -1

C = array([4, 4, 4, 0, 4])

A = array([[2, 0, 4, 1, 1]])

b = [8]

Optimal Solution is:

[1, 1, 1, 1, 1]

16

Modified Equality GIPSolver

[1. 0. 1. 1. 1.]

12.0