Solving the Knapsack problem by an advanced iterative method

DAMIAN MALARCZYK

24.12.2016

1. Problem Statement

For Knapsack with W weight capacity, within n items (x_i, $i \leq n$), where each has given weight (x_{iw}) and cost (x_{ic}), find such combination of them, that ($x_{1w} + x_{2w} + \ldots + x_{nw} \leq W$) and ($x_{1c} + x_{2c} + \ldots + x_{nc} = MAX$)

2. Possible solutions

• Genetic algorithm

Based on the idea of evolution, where an individual represents a sample solution, through individuals selection, their crossover and mutation in order to create new ones and iterative generations of whole population some solutions can be found. Finding optimal solution isn't guaranteed.

3. Used algorithms

• Genetic programming

• Knapsack genetic scheme

Knapsack solution as an individual in evolution algorithm is present as structure which wraps the result itself wrapped in integer type and bit encoded (*s*), contains fitness and evaluation values, cost as of current solution, also it stores number of all the items that could be put in the knapsack (*n*).

Crossover - with uniform distribution random amount of points to cross on is chosen, with minimum being one and maximum being *n / 4*, found points divide *s* into sectors which are then exchange between two solutions.

Mutation - 4 underlaying methods are used in my algorithm, namely:

- Adjacent swap - exchange of values between random bit and it's neighbor
- Random swap - exchange of values between two random bits
- Replacement - negation of random bit
- Removal - setting random bit to 0

Methods are chosen with Gaussian distribution using Box-Muller transformation.

• Evolution algorithm

- Definitions

  having p as **population**:

  p.n - amount of individuals in population

  having x as an **individual**:

  x.e - **evaluation** value, absolute difference between x's objective and best possible objective

  x.fit - **fitness** value, *x.e* divided by average of evaluation values within whole population

- Input

  x - input data

  p - existing population

  n - p population size

  b - best individual within population *p*

  m - biggest fitness value within individuals in *p*

  mp - mutation probability

  cp - crossover probabilty

  elitism - number of individuals to preserve

- Output

  pn - new population

  bn - best individual within population pn

  mn - biggest fitness value within individuals in p

- Steps list

1. **if** *b.fit* = 0
    1. *pn* = *p*
    2. *bn* = *b*
    3. *mn* = *m*
    4. return *pn, bn, mn*
2. // selection: *limit* <- *p.n* - *elitism*
    1. *sp* <- empty population
    2. **for each** individual(*i*) in *p*
        1. *max* <- *m* - *b.fit*
        2. **if** *max* = 0
            1. *max* <- 1
        3. *d* <- random number <0,1>
        4. **if** *d* >= (*i.fit* - *b.fit*)
            1. **append** *i* to *sp*
        5. **if** *sp.n* == *limit*
            1. **break**

            …

3. // having selected *sp.n* individuals, crossover will be used to expand new population to *p.n* - *elitism* size
    1. index <- 0
    2. while *pn.n* < *p.n* - *1*
        1. *dad, mum* <- choose two individuals from *sp*
        2. d <- random number <0,1>
        3. if *d* <= *cp*
            1. then
                1. son, daughter <- **crossover** *dad, mum:* create two new ones
                2. evaluate absolute cost of new *son*, *daughter*
                3. **replace** *dad* with *son*, if *son* is a proper solution
                4. **replace** *mum* with *daughter*. if *daughter* is a proper solution
        4. **add** dad, mum to pn
4. **for each** individual(*i*) in *pn*
    1. *d* <- random number <0,1>
    2. if d <= mp
        1. j <- **mutated** i
        2. if j is a proper solution
            1. **evalute** *j.e*
            2. **replace** *i* **in** *pn* with *j*
5. **add** `*elitism*` number of elements from *p* to *pn*
6. *avg* <- sum(*pn[].e*) / *pn.n*
7. **for each** individual(*i*) in *pn*
    1. *i.fit* <- (*i.e* / *avg*)
8. *bn* <- individual in *pn* with lowest fitness value
9. *mn* <- individual's fitness in *pn* with highest fitness value
10. return *pn*, *bn*, *mn*

4. Results

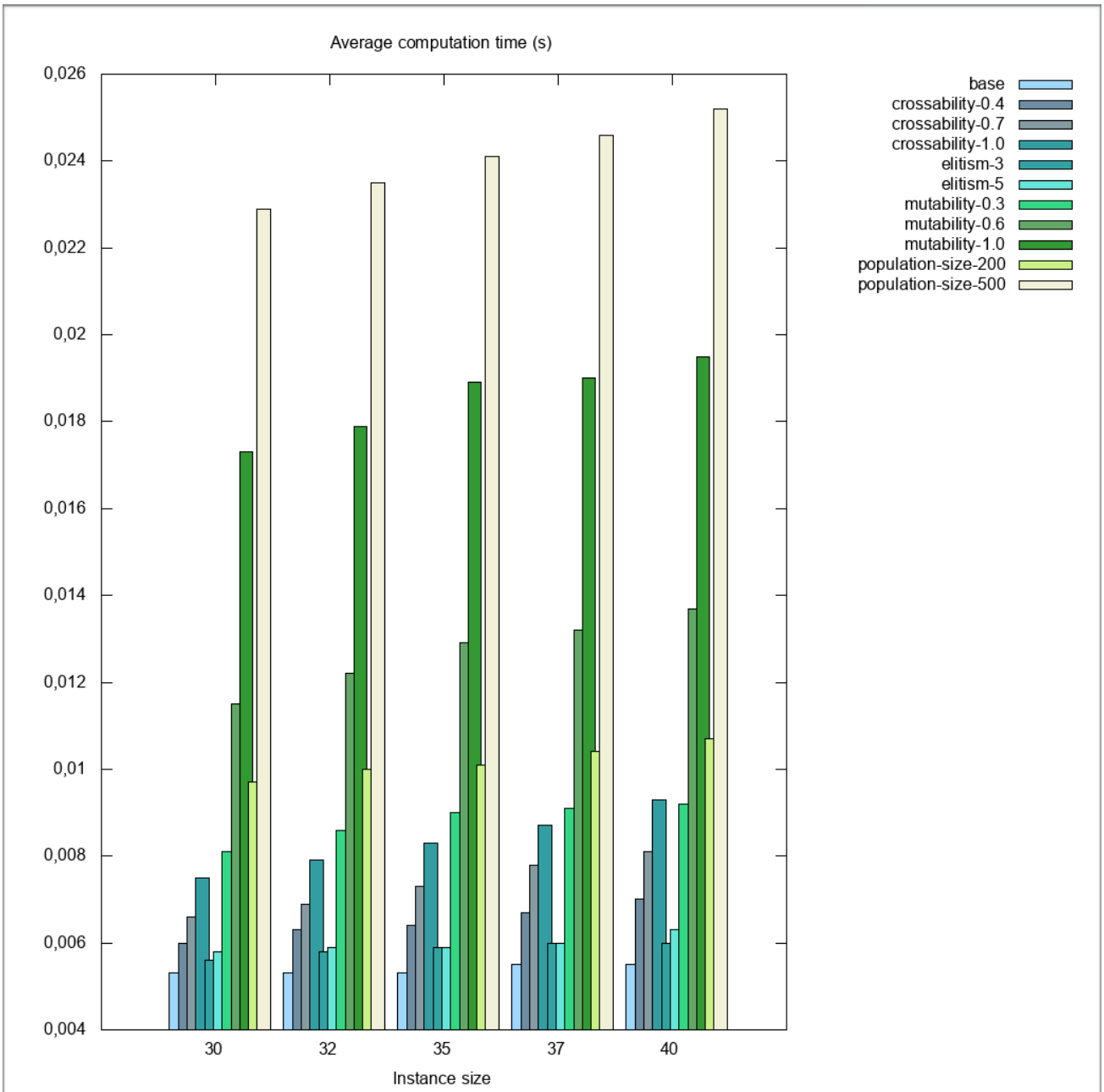
• Testing machine

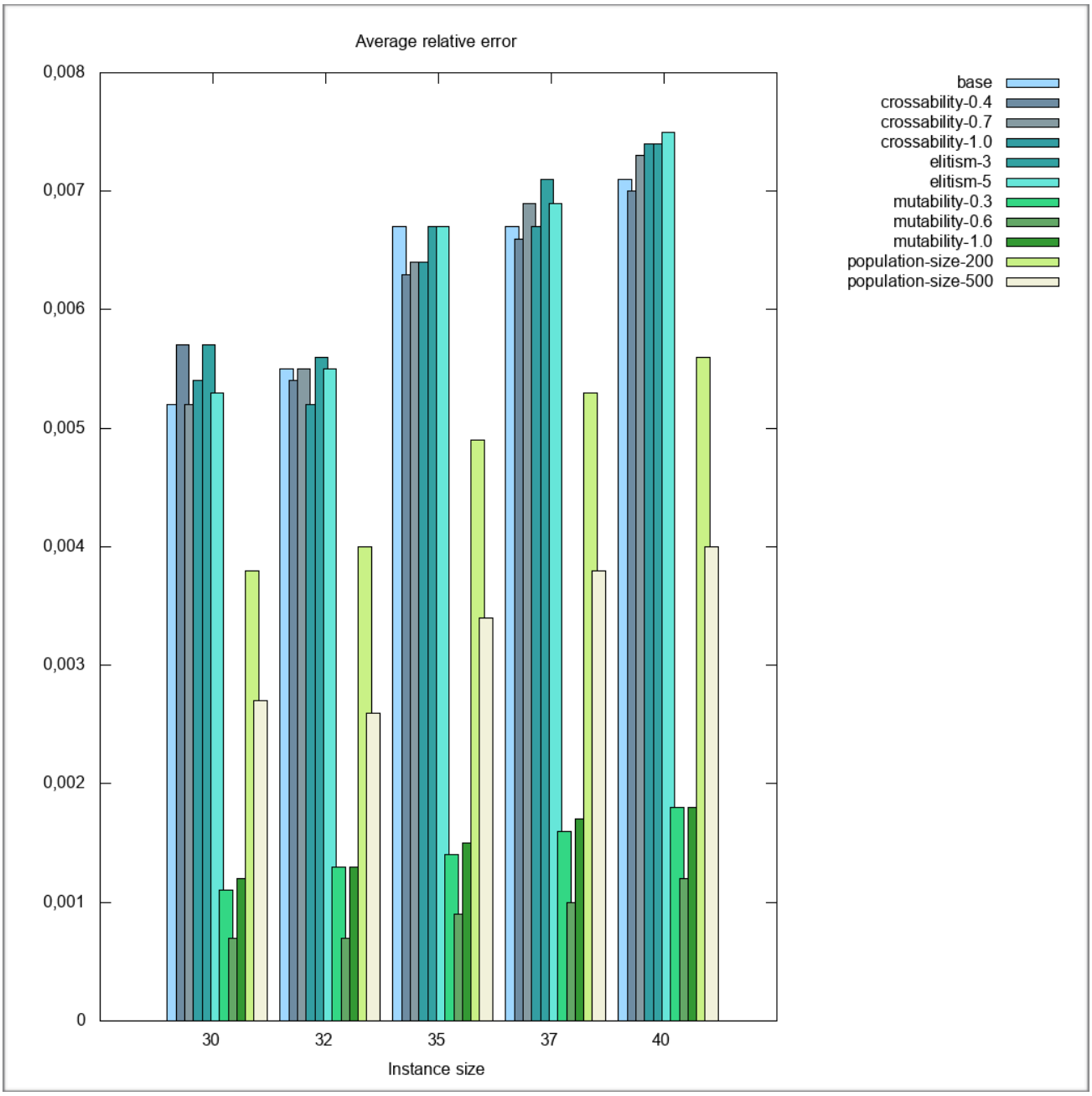
Tests were run on MacBook Air with macOS 10.12.1, Intel Core i5 4th gen 1.4 GHz CPU, 4 GB RAM 1600 Mhz DDR3, SSD hard disk model: APPLE SSD SD0128F.

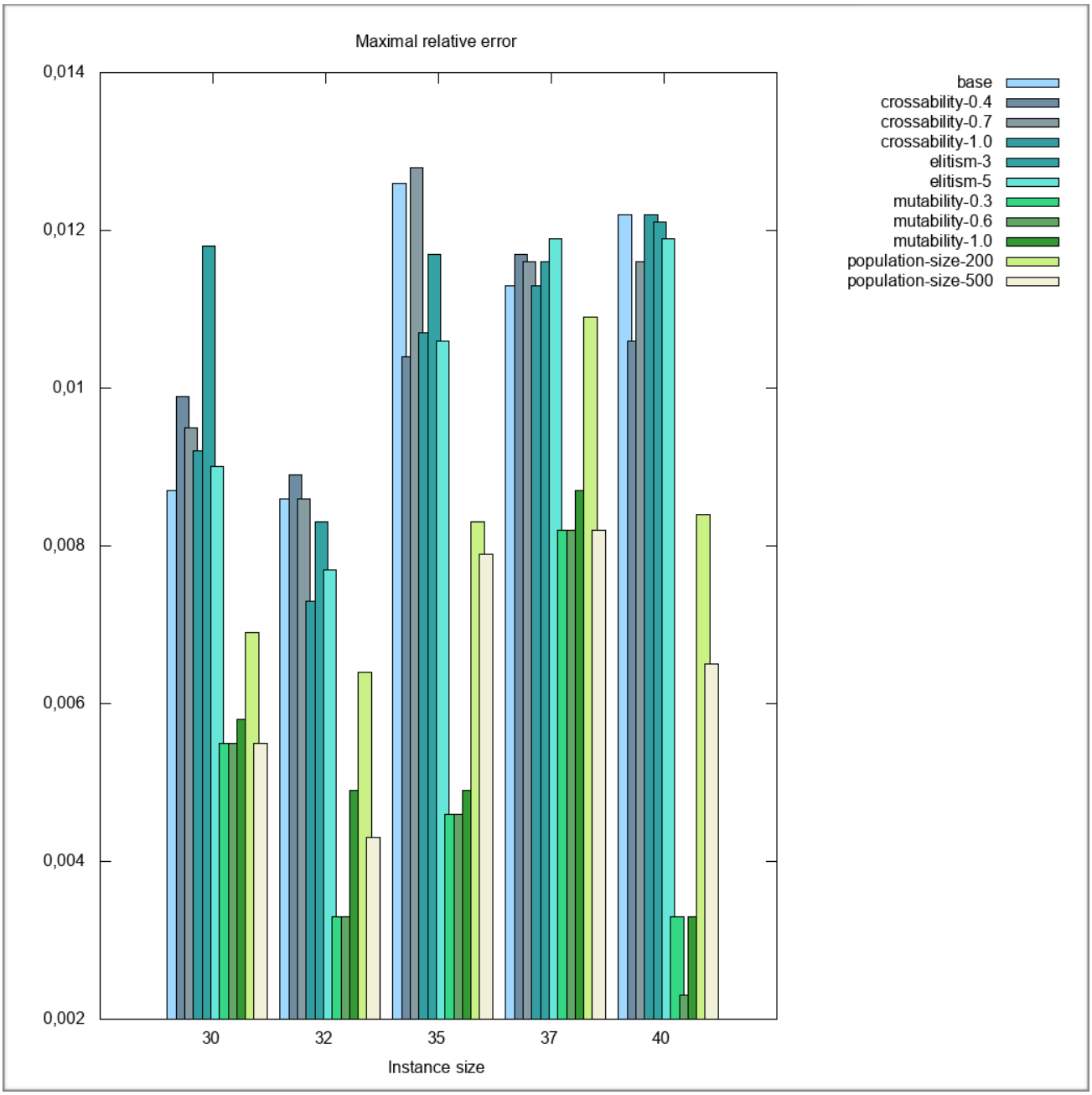
• Parameters impact

Base configuration:

Population size - 100
Mutability - 0.05 (probability of mutation)
Crossability - 0.1 (probability of crossover)
Elitism - 1 (number of best individuals to preserve)
Number of evolutions - 200

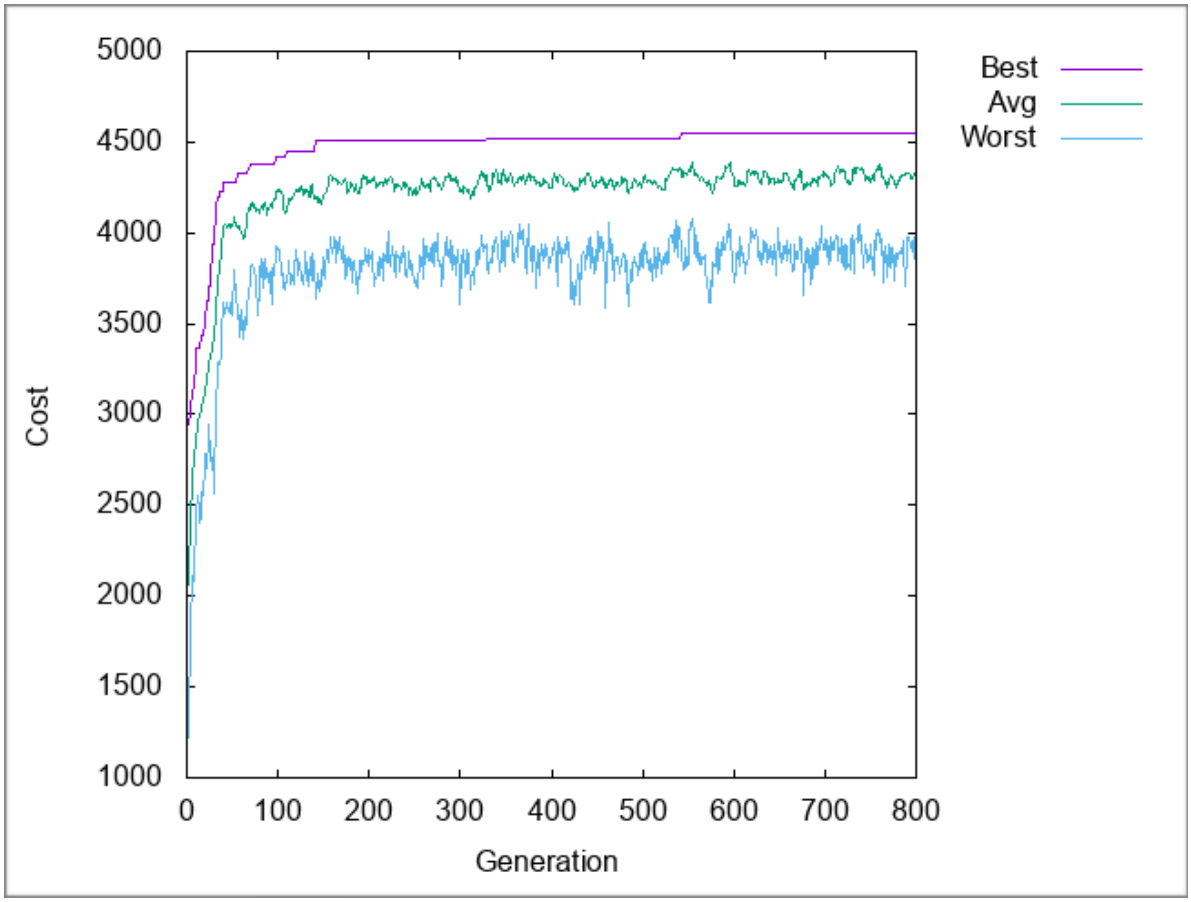


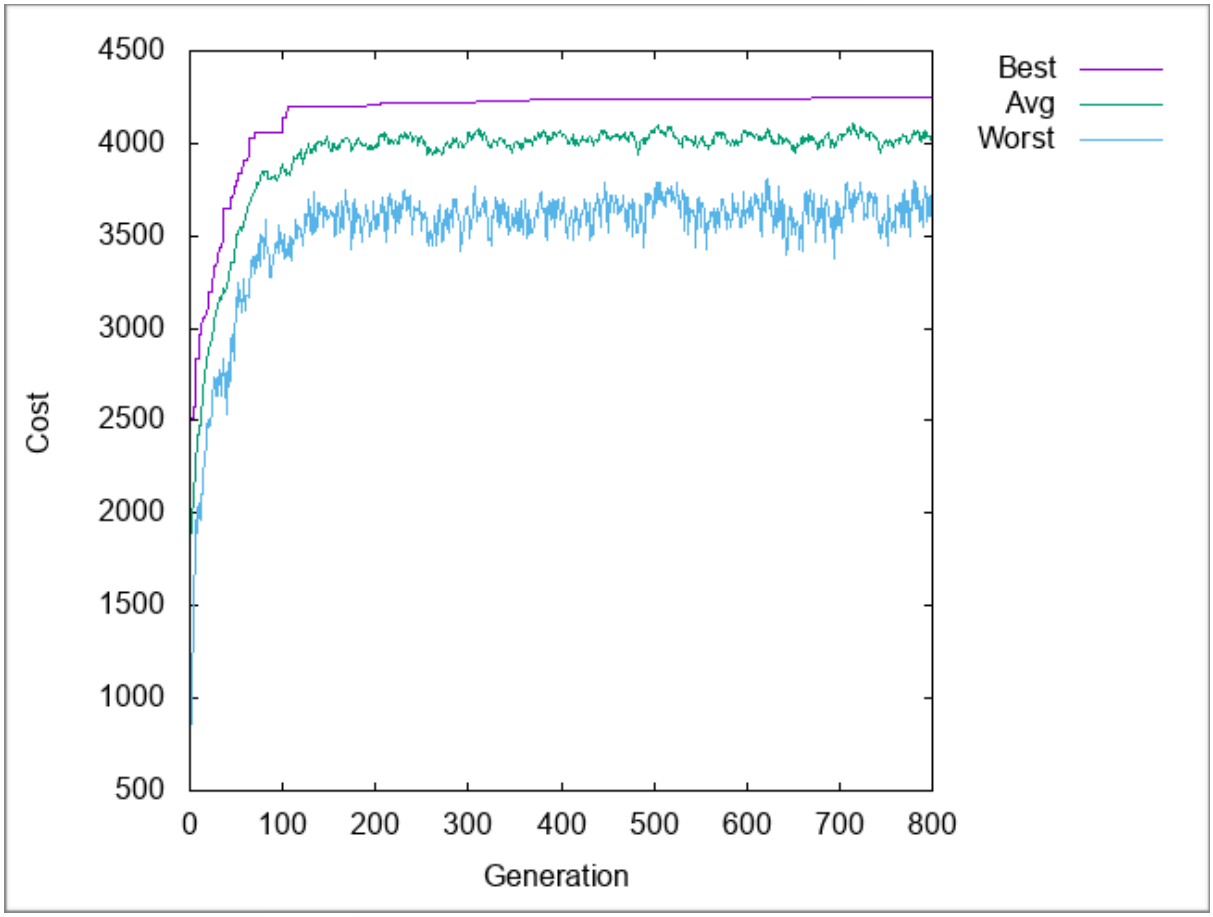

Average relative error
0,008
0,007
0,006
0,005
0,004
0,003
0,002
0,001
0
30
32
35
37
40
Instance size
base
crossability-0.4
crossability-0.7
crossability-1.0
elitism-3
elitism-5
mutability-0.3
mutability-0.6
mutability-1.0
population-size-200
population-size-500

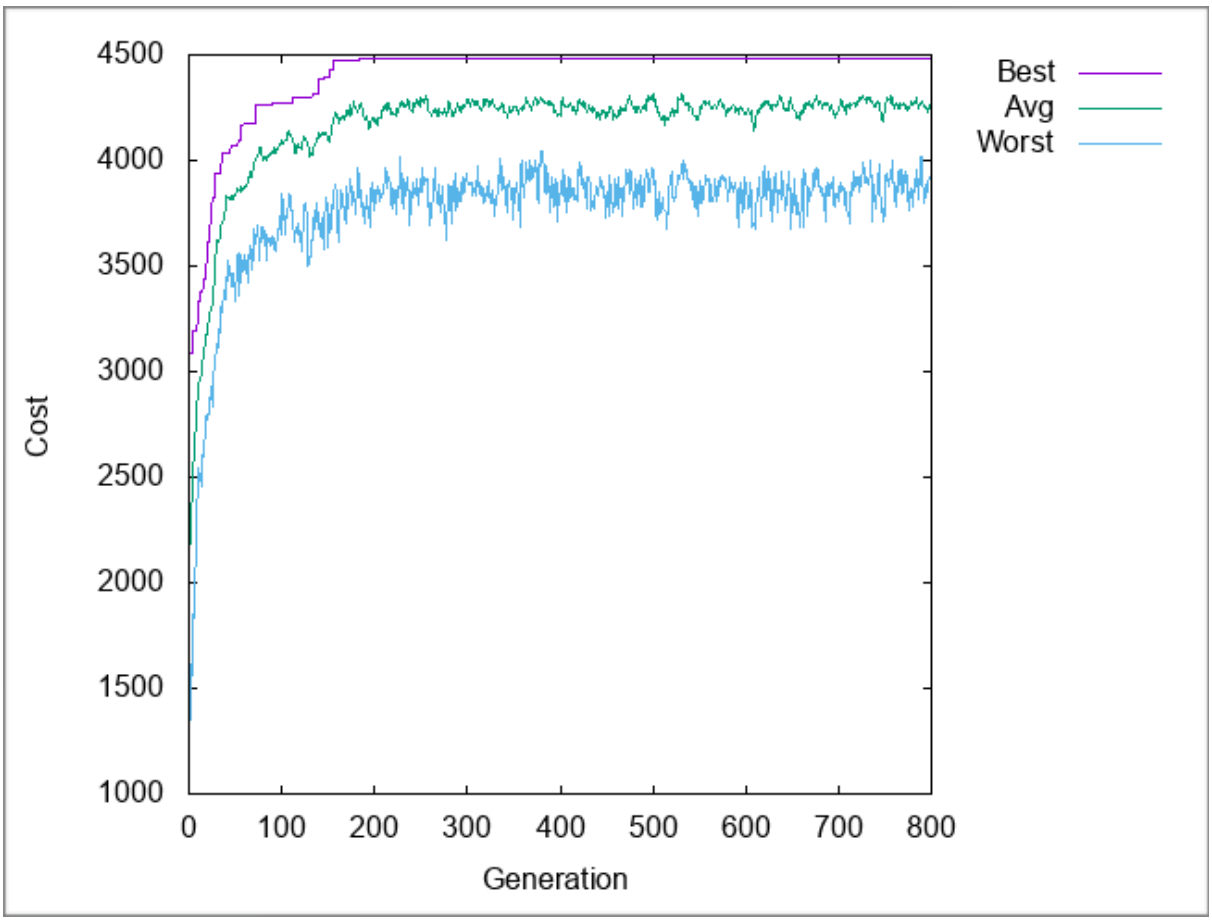

Maximal relative error
0,014
0,012
0,01
0,008
0,006
0,004
0,002
30
32
35
37
40
Instance size
base
crossability-0.4
crossability-0.7
crossability-1.0
elitism-3
elitism-5
mutability-0.3
mutability-0.6
mutability-1.0
population-size-200
population-size-500

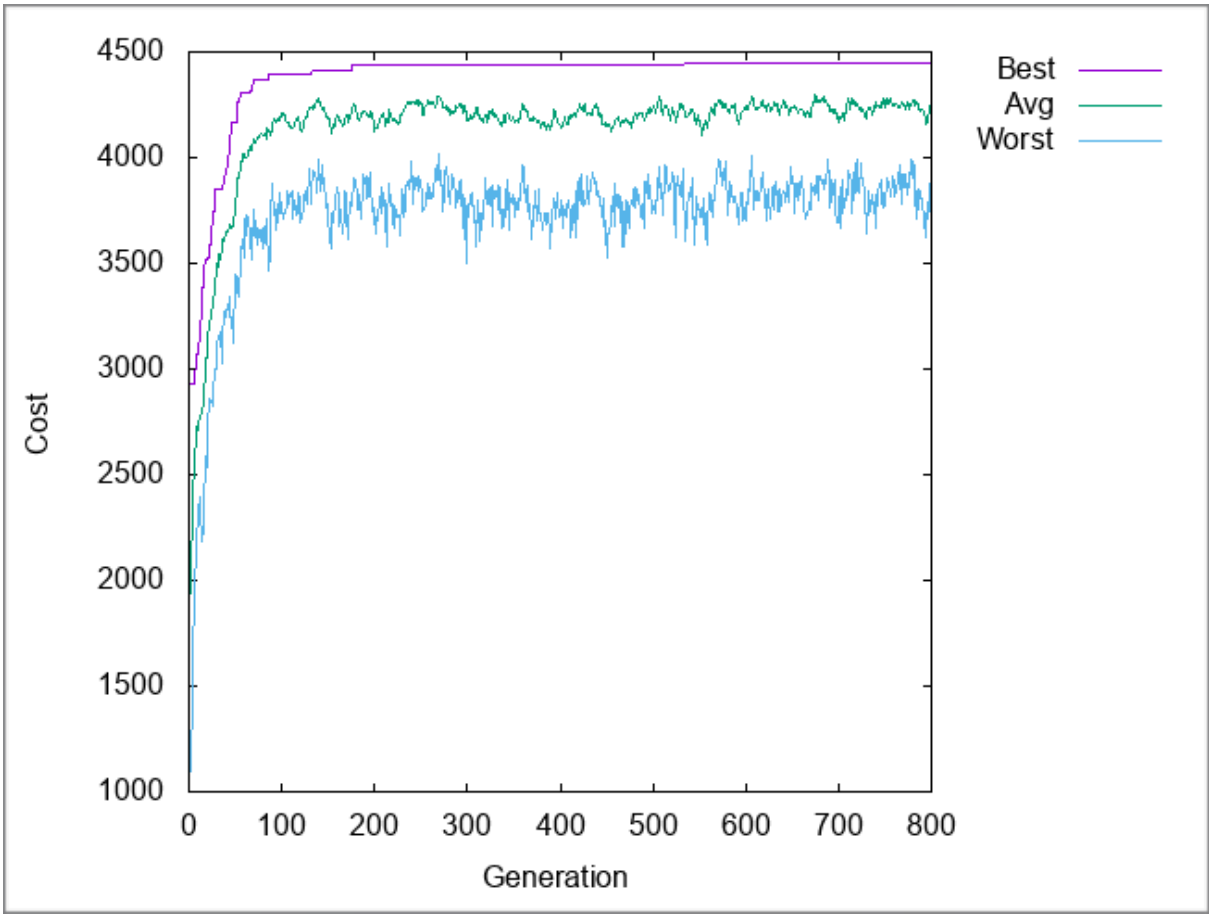
- **Iterative power**

Graphs represent algorithm progress for different instances, each of size 40 and same configuration: mutability 0.6, crossability 0.8, population of size 100.






Best
Avg
Worst
Cost
Generation
4500
4000
3500
3000
2500
2000
1500
1000
0
100
200
300
400
500
600
700
800


Best
Avg
Worst
Cost
Generation
4500
4000
3500
3000
2500
2000
1500
1000
0
100
200
300
400
500
600
700
800

5. Conclusions

As for computational time impact of parameters is pretty clear in case of genetic algorithm - rising them to higher value will increase somewhat steadily increase the computation as there will be more operations done. The change is the smallest in case of crossover probability as there is only exchange of underlaying bits and evaluation of solutions, in case of mutation parameter the increase is higher, not only solution changes are made but also earlier mentioned Box-Muller transformation is used which takes quite significant amount of time. The biggest increase is visible in case of changing population size, simply all operations have to be made for higher amount of items, thus the high increase. Slight computational time increase is notable when more than 1 elite individuals are kept, the reason for that is that when only one is used then there's no need to look for it as the algorithm constantly keeps tracks after current best individual, while willing to have more requires finding them at first.

In case of relative error of the results it seems like mutation probability has the biggest impact, though it also can be seen that setting it to the highest value does not help, value somewhere in between needs to be found. Population size of course also quite significantly impacts results, simply because there's bigger search space so more solutions can be found in fewer amount of generations. When it comes to crossover and elitism, it seems like they do not have such a big impact on their own, improvements can be seen but they are not really significant.

The algorithm without any problem managed to solve big instance problems, while algorithms finding optimal solutions failed because of their memory requirements, computational time. Sacrificing some of the time genetic algorithm was able to find better solutions than simple ratio heuristic

6. Source code link

Source

Main files can be found in Sources/Task4 folder