

Program Structure and Algorithms

Final Project

On

Knapsack Problem Using Genetic Algorithm

Index

| | |
|--|----------|
| 1) Problem description and solution explanation | 2 |
| 2) Input Result | 3 |
| 3) Graph Result | 4 |
| 4) Test Case Results | 5 |

Program Structure and Algorithms

To implement Genetic Algorithm to solve Knapsack problem, I have created 2 classes: InputUser class and Knapsack class.

InputUser class mainly focusses on input from the user which is provided to the Knapsack class. The Knapsack class mainly focusses on implementation of Genetic algorithm which solves the problem of knapsack and gives the best item list to be included in knapsack in the end. It has different methods such as startOver, firstGeneration, nextGeneration, evolutionOfBreed, finalResult which are used for generating initial and next generation, calculation of fitness for each generation and then crossover/cloning depending upon random generated value.

The Genetic algorithms are mainly designed for solving the problem with Survival of the fittest strategy which evolved in stages and we get the best fitness outcome in the end.

In the InputUser class, I have taken different values from the user as an input from the console and the same is provided to the Knapsack class.

- Total number of items is an important parameter for the knapsack problem as all the calculations will be derived based on total items, their weighs and values.
- Knapsack capacity is main deciding parameter cause sum of weights of all items can not be more than the knapsack capacity, at the same time their value should be as large as possible to get them selected.
- The parameter total population size decides how many times the generations or breeds are getting created after very first generation which is exclusive.
- Crossover probability and mutation probability are taken into consideration for generating next breeds to test the fitness for knapsack problem.

In the Knapsack Class, I have defined numerous methods as mentioned earlier which are responsible for evolution and calculation of next breeds.

- The main method gets the input populated into object of the class and then the object calls a method startOver which is responsible for very first generation which is based on random generation of values. First gene set is then populated with values of '0' and '1' and each chromosome of every gene gets populated with '0' or '1'.
- Then I have evaluated the fitness of every gene in the first generation by calculating the total weight of the gene array.
- Then I have checked the best result from first generation gene set to check which is the best fir for the solution of the problem. At the same time, calculating the mean of fitness array for further analysis.
- The next step is to breed the generation and check for further results which can be used as best fit for the solution of knapsack problem (Stopping condition is checked).
- The genes are selected on the random basis for breeding next generation.
- The crossover and cloning are the two techniques used for breeding the next generation which are selected on the random basis and user input.
- Mutation on the next generated breed is dependent on the probability of mutation which is taken from user and some random number.

Program Structure and Algorithms

- The same procedure is applied to the next generations which has been applied for the very first generation and calculated the results for best fits.
- In the end, optimal results are calculated and printed so that the perfect solutions which can be used for filling the knapsack are listed.

The graph generated for mean of fitness values and best fit results of all the generations for following input is as followed:


Please enter total capacity of knapsack 300

Please enter total population size 100

Please enter total number of Generations 100

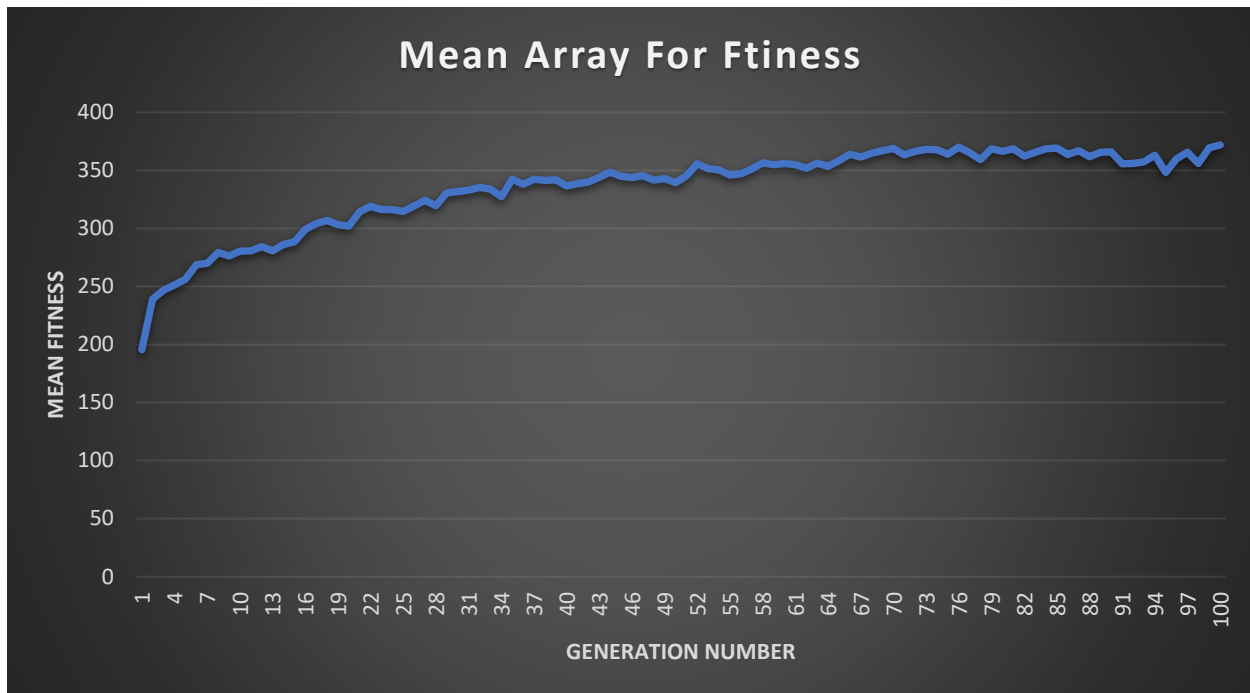
Please Enter prob for cross: 0.5

Please enter prob for mutation: 0.03



```
run:
Enter total number of items for knapsack:
20
Please give input for Value of Item 1:
1
Please give input for weight of Item 1:
20
Please give input for Value of Item 2:
2
Please give input for weight of Item 2:
29
Please give input for Value of Item 3:
3
Please give input for weight of Item 3:
28
Please give input for Value of Item 4:
4
Please give input for weight of Item 4:
27
Please give input for Value of Item 5:
5
Please give input for weight of Item 5:
26
Please give input for Value of Item 6:
6
Please give input for weight of Item 6:
25
Please give input for Value of Item 7:
7
Please give input for weight of Item 7:
24
Please give input for Value of Item 8:
8
Please give input for weight of Item 8:
23
```

Program Structure and Algorithms



```
Algorithm Project (run) x Algorithm Project (run) #4 x Algorithm Project (run) #5 x
89 - 403.0
90 - 385.0
91 - 379.0
92 - 386.0
93 - 388.0
94 - 396.0
95 - 345.0
96 - 371.0
97 - 392.0
98 - 339.0
99 - 386.0
100 - 404.0
Best result of next generation 100: 01001101011011101111111111111111
Mean fitness value of next generation: 371.91
Fitness value of best result of next generation 100: 413.0
Cross observed 22 times
Clone observed 28 times
Mutation did happen
Best list result of items that can be included in knapsack are:
4 5 6 8 11 12 13 14 15 17 18 19 20 21 22 23 24 25 26 27 28 29 30
Mean array for Fitness
```

Program Structure and Algorithms

The result for the test cases which are testing the result for the best fitness value is as followed:

The screenshot displays an IDE with several tabs: `InputUser.java`, `Knapsack.java`, `Output`, `TestKnapsackTest.java`, and `Test Results`. The active tab is `Test Results`, which shows the following test results:

- Tests passed: 100.00 %
- All 6 tests passed. (0.208 s)
- knapsackProblem.TestKnapsackTest passed
 - test1 passed (0.079 s)
 - test2 passed (0.001 s)
 - test3 passed (0.001 s)
 - test4 passed (0.0 s)
 - test5 passed (0.002 s)
 - test6 passed (0.0 s)

The `Output` tab shows the following program output:

```
Populate Value:
1 - 01111
2 - 01111
3 - 01111
4 - 01111
5 - 01011
6 - 11111
7 - 01111
8 - 01111
9 - 01111
10 - 01111

Next Generation Fitness Value:
1 - 14.0
2 - 14.0
3 - 14.0
4 - 14.0
5 - 11.0
6 - 15.0
7 - 14.0
8 - 14.0
9 - 14.0
10 - 14.0

Best result of next generation 8: 11111
Mean fitness value of next generation: 13.8
Fitness value of best result of next generation 8: 15.0
Cross observed 2 times
Clone observed 2 times
Mutation did not happen
Next Generation 9:
*****
Populate Value:
1 - 01011
2 - 01111
3 - 01111
4 - 01111
5 - 01111
6 - 01111
7 - 01111
8 - 01111
9 - 01111
10 - 01111

Next Generation Fitness Value:
1 - 11.0
2 - 14.0
3 - 14.0
4 - 14.0
5 - 14.0
6 - 14.0
7 - 14.0
8 - 14.0
9 - 14.0
10 - 14.0

Best result of next generation 9: 01111
Mean fitness value of next generation: 13.7
Fitness value of best result of next generation 9: 14.0
Cross observed 2 times
Clone observed 2 times
```