A Genetic Algorithm Application for Knapsack Problem in Java

Author: Ahmet Kaşif
E-Mail: ksfahmet@gmail.com

Foreword

The project is built on top of a base application[5] implemented in Java. The differences in implementation are shown below:

1. Fitness function of referred application was to converge 64 bit binary numbers to a pre-built chromosome. Our problem's aim is to find optimal item combination which satisfy knapsack weight limit.
2. Elitism approach has been used in referred application, while we did not used it because the experimentations did not give good results.
3. Referred application used a number of iterations to stop, while it is updated with a maximum number of non-improving solutions implementation.

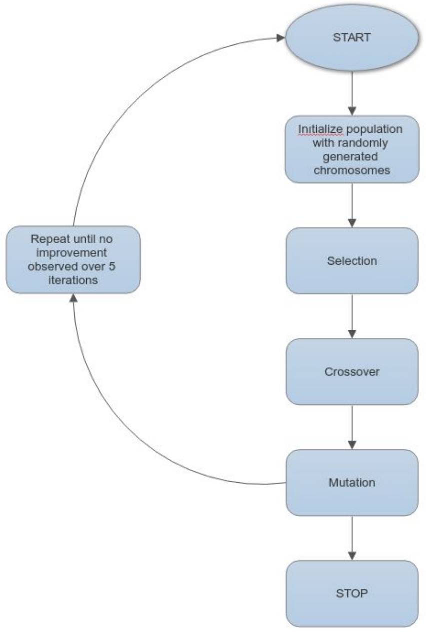
Knapsack Problem

Knapsack problem is one of the most famous NP-Hard problems. [3]

Given set of items, each with a weight and value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

Application areas of this problem are generally resource allocation problems, which can be an economics problem, cryptography problem, computer science problem etc.

Algorithm


START
Initialize population with randomly generated chromosomes
Selection
Crossover
Mutation
STOP
Repeat until no improvement observed over 5 iterations

Chromosome encoding

Chromosomes are encoded using a byte array with a length of item size. Each byte (gene) represents if an item is selected or not.

(0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0)

Fitness Function

Fitness function is summation of values of items selected in the chromosome. If weight of the chromosome is above knapsack capacity, we update fitness level of that chromosome with -1. Problem is designed as a maximization problem, so bigger fitness means better.

F = ∑ values, if weight < knapsack capacity

F = -1, otherwise

```
fitness ← 0
weight ← 0
i ← 0

while i < chromosome size
    if i. gene's value is 1
        fitness ← fitness + value of i. item
        weight ← weight + weight of i. item

if weight - knapsack capacity > 0
    fitness ← -1

return fitness
```

Algorithm and Dataset

A sample dataset[2] with its own optimal value and knapsack capacity has been used in this application, and it has been tried to converge to that optimal value while still keeping runtime fast.

```
population ← generatePopulation
sc ← 2 // max no-improvement iteration count

while sc > 0
    int i ← 0
    bestChromosome ← get best chromosome present in population
    while i < populationSize
        select 2 chromosomes from population using tournament selection method
        apply crossover to selected chromosomes
        apply mutation to crossover applied new chromosome
        pass new chromosome into next generation

    if bestChromosome fitness value = newPopulation’s best chromosome’s fitness value
        sc ← sc -1
    else
        sc ← 2
    if bestChromosome fitness value < newPopulation’s best chromosome’s fitness value
        bestChromosome ← newPopulation’s best chromosome
```

Generating a Chromosome

Chromosome generation is a random process which can produce chromosomes not satisfying knapsack limit rule. In order to overcome this problem, it is repeatedly started over until a feasible chromosome is produced. Below you can see the pseudocode of generation of one chromosome:

```
weight ← knapsackLimit
while weight <= 0 // Repeat until a positive weight returns meaning we do not exceed knapsack limit
    weight ← knapsackLimit
    for i ←0 & i < chromosomeSize
        gene ← randomly assign 0 or 1
        weight ← weight - weights[i] // Subtract added item's weight from limit
        i ← i + 1
```

Tournament Select Implementation

Tournament select has been implemented with tournament size of 5. Random 5 chromosome is selected from population and best of them is selected. It’s pseudocode is as follows:

```
population ← dedicate memory with a population of size tournamentSize
for i ←0 & i < tournamentSize
        add random chromosome from population to chromosome
        i ← i + 1
return best chromosome of these 5
```

Crossover Implementation

Crossover pseudocode can be seen below:

```
newChromosome ← dedicate memory for newChromosome
for all genes of two chromosomes
    if random number > crossover rate
        add chromosome1's gene to newChromosome
    else
        add chromosome2's gene to newChromosome
    i ← i + 1
return newChromosome
```

Mutation Implementation

Mutation pseudocode can be seen below:

```
for all genes of two chromosomes
    if random number < mutation rate
        mutate chromosome gene by a new random value
    $i \leftarrow i + 1$
```

Deciding on GA parameters

After implementing a basic running GA, next step is to determine which parameter values work best for the algorithm and to decide if we need to update any approach.

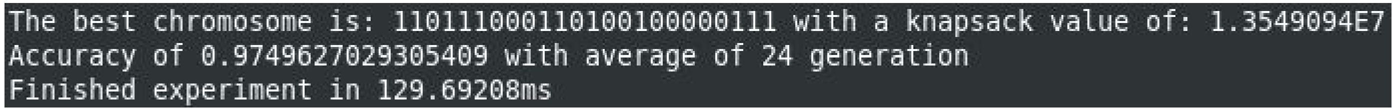
Each parameter has been decided using trial & error approach. Experiments have been made by running algorithm 100000 times and taking average of all results.

Population size, crossover rate, mutation rate, elitism presence and iteration limit of no-improvement has been tested.

Deciding on GA paremeters

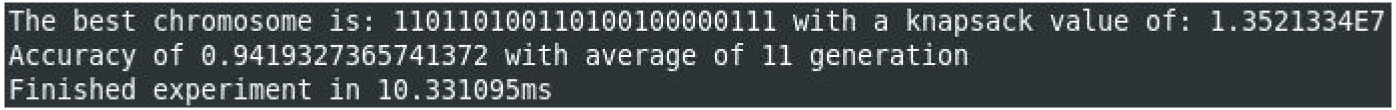
Population Size: Experimentation started from 50 and it has been seen that reducing it does not affect optimality. After some number of trials, population size has been selected as 8 while looking for best optimality and average generation count. Below, you can see the comparison of population size 50 and 8:

Results for population size: 50



```
The best chromosome is: 1101110001101001000001111 with a knapsack value of: 1.3549094E7
Accuracy of 0.9749627029305409 with average of 24 generation
Finished experiment in 129.69208ms
```

Results for population size: 8

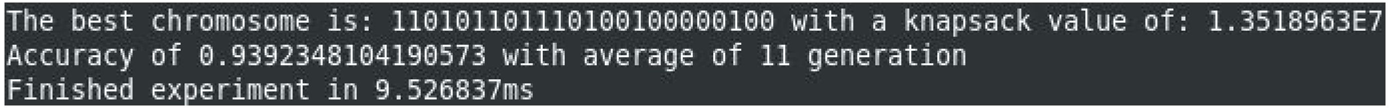


```
The best chromosome is: 1101101001101001000001111 with a knapsack value of: 1.3521334E7
Accuracy of 0.9419327365741372 with average of 11 generation
Finished experiment in 10.331095ms
```

Deciding on GA paremeters

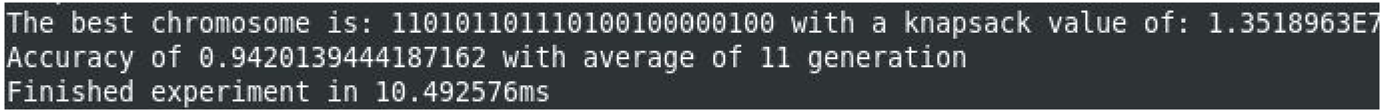
Crossover Rate: Experimentation started from 0.5 as crossover rate it has been observed that going towards 0.9, gives better results in scale of 1/1000 and while there hasn't been any gain on average generation count, running time is nearly %10 faster. Below you can see the comparison of crossover rates of 0.9 and 0.5:

Crossover rate of 0.9



```
The best chromosome is: 110101101110100100000100 with a knapsack value of: 1.3518963E7
Accuracy of 0.9392348104190573 with average of 11 generation
Finished experiment in 9.526837ms
```

Crossover rate of 0.5

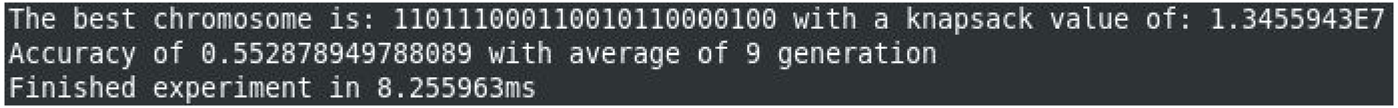


```
The best chromosome is: 110101101110100100000100 with a knapsack value of: 1.3518963E7
Accuracy of 0.9420139444187162 with average of 11 generation
Finished experiment in 10.492576ms
```

Deciding on GA paremeters

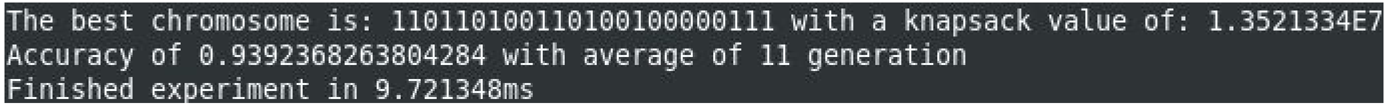
Mutation Rate: Experimentation of mutation rate started from 0.015 and it has been seen that increasing mutation rate gave much better optimality rates while not having a big impact on running time and average generation count. Below you can see the comparison of mutation rates between 0.015 and 0.064:

Mutation rate of 0.015



```
The best chromosome is: 1101110001100101100000100 with a knapsack value of: 1.3455943E7
Accuracy of 0.552878949788089 with average of 9 generation
Finished experiment in 8.255963ms
```

Mutation rate of 0.064

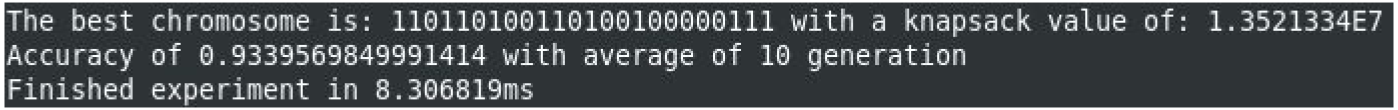


```
The best chromosome is: 1101101001101001000000111 with a knapsack value of: 1.3521334E7
Accuracy of 0.9392368263804284 with average of 11 generation
Finished experiment in 9.721348ms
```

Deciding on GA parameters

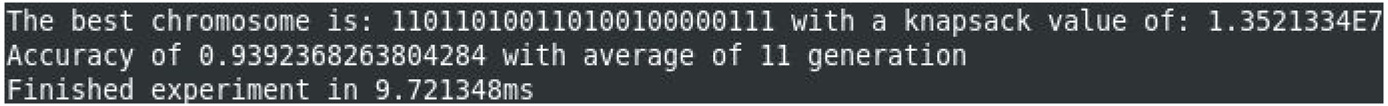
Elitism presence: It is experimented if usage of elitism gives better optimality or runtime improvement and it is observed that usage of elitism gives poor results even if it shortens runtime. Below you can see the comparison between using elitism and not:

Results using elitism



```
The best chromosome is: 110110100110100100000111 with a knapsack value of: 1.3521334E7
Accuracy of 0.9339569849991414 with average of 10 generation
Finished experiment in 8.306819ms
```

Results not using elitism

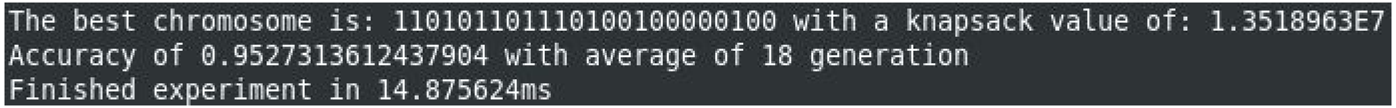


```
The best chromosome is: 110110100110100100000111 with a knapsack value of: 1.3521334E7
Accuracy of 0.9392368263804284 with average of 11 generation
Finished experiment in 9.721348ms
```

Deciding on GA parameters

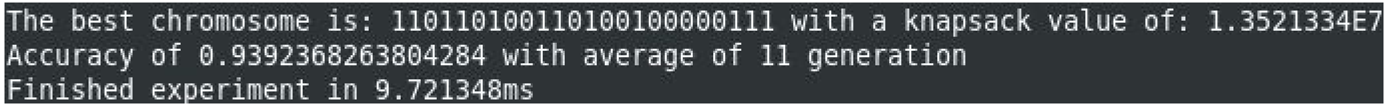
Stop Criteria: Maximum number of no-improvement iteration count has been selected as stop criteria instead of a more standard iteration count approach, because our policy to keep average generation count at check and increase runtime efficiency also. After implementation is done, we started from 5 as our maximum. Below you can see the comparison between iteration count 5 and 2:

Maximum no-improvement iteration as 5



```
The best chromosome is: 1101011011101001000000100 with a knapsack value of: 1.3518963E7
Accuracy of 0.9527313612437904 with average of 18 generation
Finished experiment in 14.875624ms
```

Maximum no-improvement iteration as 2



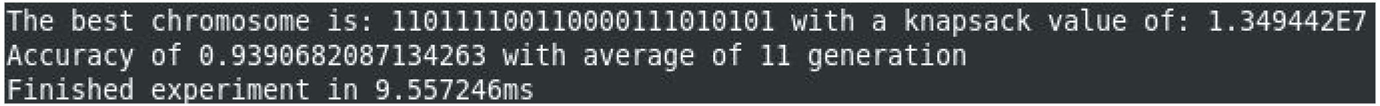
```
The best chromosome is: 1101110100110100100000111 with a knapsack value of: 1.3521334E7
Accuracy of 0.9392368263804284 with average of 11 generation
Finished experiment in 9.721348ms
```

Results

We have converged our results to %93.91 of optimal value present in the sample dataset as can be seen below. While better solutions could be obtained using bigger population sizes and increasing maximum no-improvement iteration limit, we have also kept runtime in check.

According to these solutions, it can be said that one can improve optimality while sacrificing from runtime efficiency or make application run faster while sacrificing optimality.

Below you can see the result of optimality and average generation count obtained using the parameters selected before:



```
The best chromosome is: 110111100110000111010101 with a knapsack value of: 1.349442E7
Accuracy of 0.9390682087134263 with average of 11 generation
Finished experiment in 9.557246ms
```

Environment

Application is implemented in Java language. Application is tested on a linux machine with a processor of 3.5 GHz of clock rate. It should also be repeated that, while time observed in milliseconds is subject to change in a more powerful machine, average generation count gives a machine-free point of view.

References

1. Knapsack Implementation using GA, http://www.dataminingapps.com/2017/03/solving-the-knapsack-problem-with-a-simple-genetic-algorithm/
2. Dataset 8, https://people.sc.fsu.edu/~jburkardt/datasets/knapsack_01/knapsack_01.html
3. Knapsack problem, https://en.wikipedia.org/wiki/Knapsack_problem#Computational_complexity
4. http://www.sc.ehu.es/ccwbayes/docencia/kzmm/files/AG-knapsack.pdf
5. Base application, http://www.theprojectspot.com/tutorial-post/creating-a-genetic-algorithm-for-beginners/3