

Computer Science Extended Essay

TOPIC

Investigating the Performance of Different Selection Strategies of
Genetic Algorithm

RESEARCH QUESTION

To what extent is the performance of tournament selection strategy better than that of roulette wheel selection in solving the Knapsack Problem in terms of convergence rate and quality of the solution with different configurations?

Word count: 3989 words

CS EE World
<https://cseeworld.wixsite.com/home>
27/34 (A)
May 2022

Donator Info:
Name: Lam Ho
Accepted into: Dual Degree between Tel Aviv University and Columbia University
"Feel free to contact me at lamho [dot] ghis [at] gmail [dot] com for any questions regarding the IB in general or the CS EE specifically, I would be more than happy to help (if I could)!"

TABLE OF CONTENT

1. Introduction	2
2. Theory	4
2.1. Genetic algorithm.....	4
Exploitation and Exploration	
Crossover	
Mutation	
Premature convergence	
Elitism	
Termination condition	
2.2. Selection strategies.....	9
2.3. The Knapsack Problem.....	11
3. Hypothesis	13
4. Methodology	14
4.1. The experimental procedure.....	14
4.2. Independent variables.....	15
Dataset	
Other parameters	
4.3. Dependent variables.....	16
4.4. Controlled variables.....	17
4.5. Efficiency measure.....	18
5. Experiment Results and Analysis	19
6. Conclusion	23
7. Bibliography	25
8. Appendix	28

1. Introduction

The field of science has witnessed many great inventions inspired by bionics, i.e the application of biological principles to the study and design of human systems (Yu et al., 2013). The submarine, for example, is an invention that mimics fish. Similarly, evolutionary algorithms (EAs) are algorithms that utilize evolutionary principles (survival of the fittest) to build adaptive systems in order to solve complex optimization problems that normally cannot be solved by deterministic algorithms (Yu et al., 2013).

Genetic algorithm, a metaheuristic pioneered by John Holland in the 1970s, is perhaps the most well-known among different types of evolutionary algorithms such as evolutionary programming, evolution strategies and genetic programming (Dasgupta & Michalewicz, 1997). It has been applied in various fields such as pattern recognition, robotics, artificial life, experts system, electronic and electrical field, cellular automata, etc (Dasgupta & Michalewicz, 1997). As a part of the larger class of evolutionary algorithms, the genetic algorithm also mimics the process of natural selection to solve optimization and search problems based on biological operators such as crossover, mutation and selection. A typical genetic algorithm consists of the following steps: initialization, evaluation, selection, crossover and mutation. Depending on the problems, there are several approaches that can be used for each step of the algorithm.

The Knapsack problem was pioneered by Dantzig in the late 1950s, opening a great number of extensive and intensive research later on in this field (Badiru, 1970). The problem exemplifies a real-life situation where we have to assign a set of items into a knapsack or a number of knapsacks in which each item has different sizes and values while the knapsack has a limited capacity. Our goal is to maximize the total value of the items without exceeding

the capacity(s) of the knapsack(s). The knapsack problem is classified as an NP-hard problem whose solutions cannot be obtained by the application of polynomial-time algorithms (Badiru, 1970). However, thanks to years of research done by scientists have presented several approaches that can be used to easily solve this problem such as dynamic programming, recursive approach, greedy algorithm, and genetic algorithm.

This paper aims to investigate the application of the genetic algorithm to the knapsack problem, specifically evaluating the performance of the two different selection strategies used: roulette wheel selection and tournament selection with different parameters. The paper will also carry out experiments with and without elitism - an algorithm that preserves the best individuals to the next generations to observe whether this factor would affect the performance of the algorithm in proposing the optimal solution or not.

This research could be proved helpful in presenting a more optimal approach when utilizing the genetic algorithm to solve the knapsack problem. The problem has a plethora of real-life applications that require computer processing allocations in distributed systems such as financial modelling, production and inventory management systems, design of queuing network models in manufacturing and last but not least, control of traffic overload in telecommunication systems (Badiru, 1970). With a faster performance in solving the problem of traffic overload, for example, it will help to prioritize different data that need to be transferred in the network with a scarcely available bandwidth; hence, improving productivity and saving lots of money in various fields.

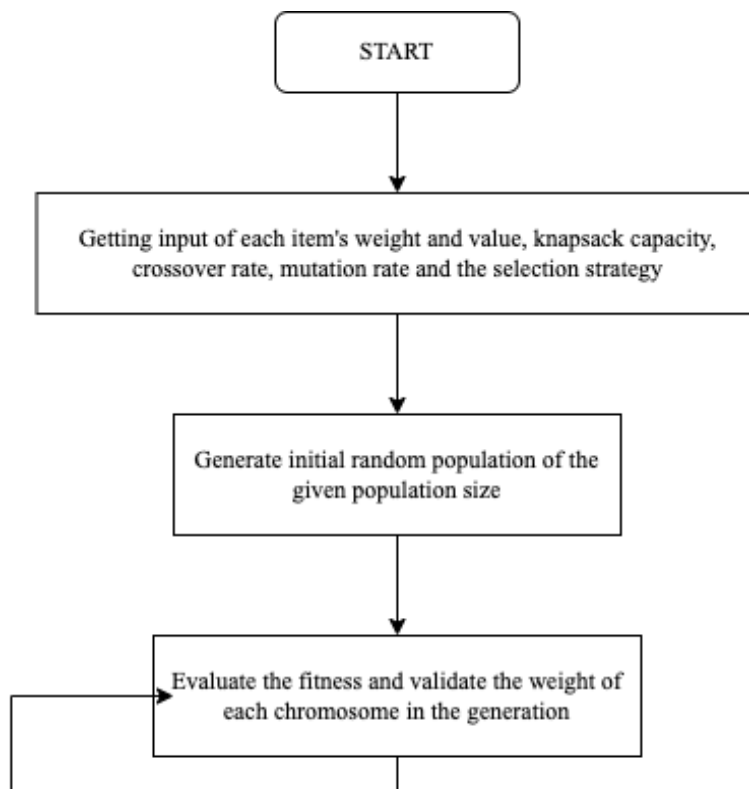
To evaluate the performance of the two selection strategies used, an experiment would be carried out to calculate how many generations it takes for each strategy to find out the

optimal solution and how close it is compared to the best solution, given that the termination condition is the same. The essay would also consider the impact of the crossover rate, mutation rate and elitism to see to what extent these variables influence the performance of each strategy.

2. Theory

2.1. Genetic algorithm

Genetic algorithm is a heuristic algorithm that is used to solve optimization problems in computational mathematics (Pan & Zhang, 2018). The algorithm applies “Darwinian principles of survival” to its operation, also known as “the survival of the fittest”. The fitter individuals will have a higher chance to adapt to the environment and therefore survive and reproduce new generations that are more endurable to nature. Similarly, using this principle, the genetic algorithm consists of three genetic operations: selection, crossover and mutation (Zhong et al., 2006). The process is illustrated by the flowchart below:



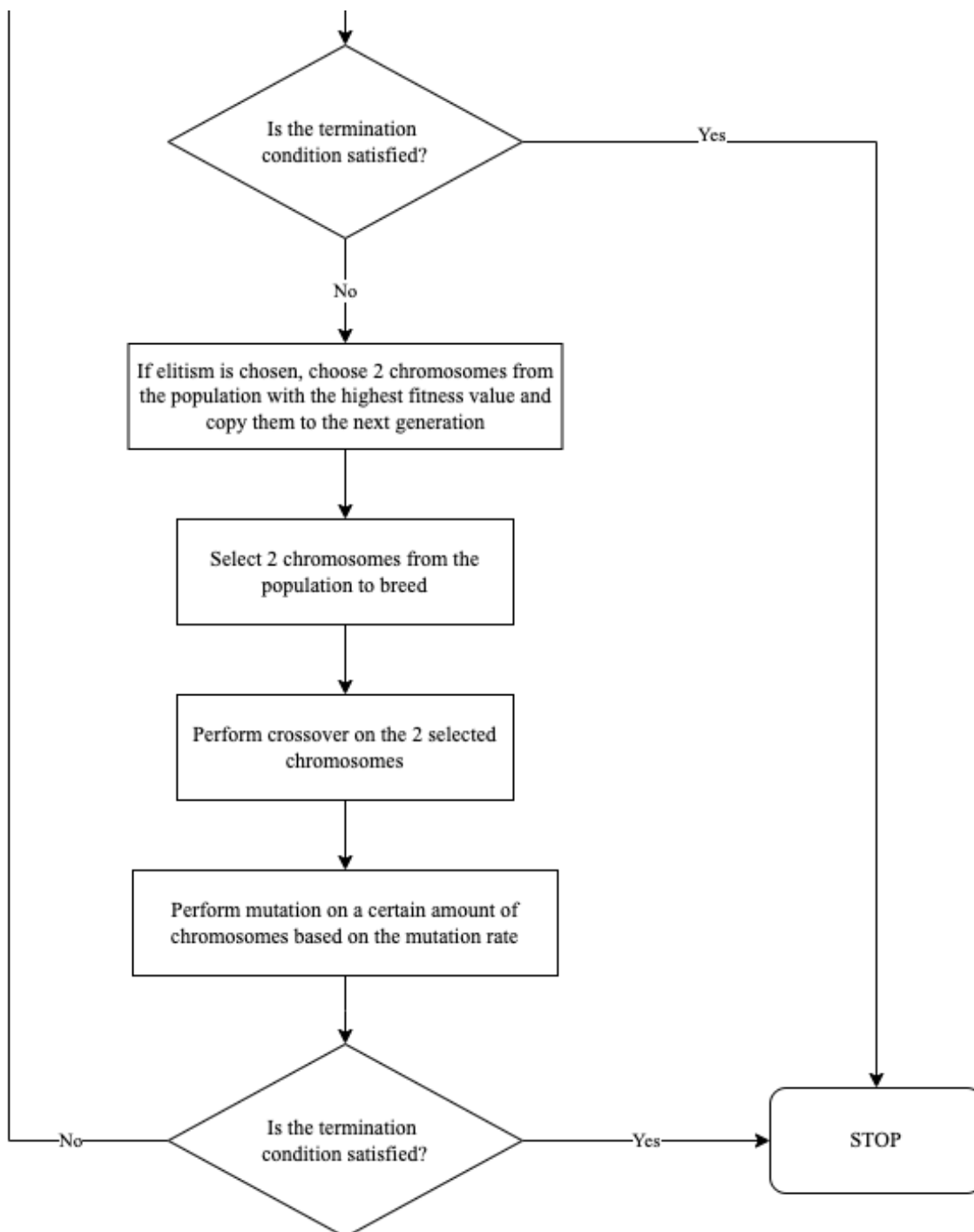


Figure 1: Genetic algorithm flowchart

2.1.1. Exploitation and Exploration

Exploitation (also called intensification) characterizes the extent to which the algorithm preserves the properties of the fittest solutions in the population. An algorithm with a high rate of exploitation will move towards the most promising areas of the search space around the best solutions found so far (Hao & Solnon, 2019). In figure 2, the population fitness will gradually gather in one of the peaks. However, as illustrated in the same figure, there are various peaks that can exist in one search space. The tallest peak (both positively and negatively) is called the global optima while the lower ones are local optima. Due to this factor, high exploitation might lead the population to be stuck in one of the local optima; thus, the result found will not be the best solution possible.

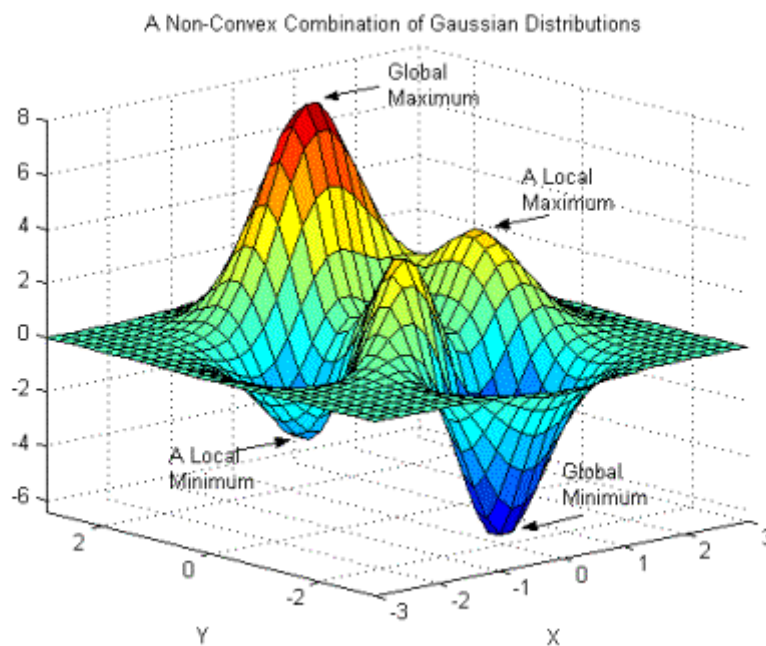


Figure 2: Illustration of a search space (source: Zachary Kaplan)

Contrastingly, exploration (also called diversification) highlights the diversity of the population. It aims at expanding the search space to discover new areas that may have better solutions. By doing this, the algorithm will avoid being stuck in the local optima and hence,

have a higher chance of reaching the global optima. However, a high rate of exploration would lead to a scattered population that cannot converge, which is also not a desired outcome that we want to achieve.

In short, exploitation and exploration play a crucial role in every search algorithm. A successful search algorithm requires a good ratio between exploration and exploitation (Črepinšek et al., 2013). This is achieved by modifying the parameters of the algorithm such as crossover rate, mutation rate, population size, etc, which will be investigated in the experimenting process.

2.1.2. Crossover

Crossover is the process of mixing bits of two chromosomes to create an offspring that has the genotype of both parents for the next generation. Firstly, it randomly chooses a locus on the chromosome, then it exchanges the subsequences before and after that locus to create the offspring (Hristakeva & Shrestha, 2022). Crossover increases the diversity of the population, therefore increasing the exploration rate of the algorithm.

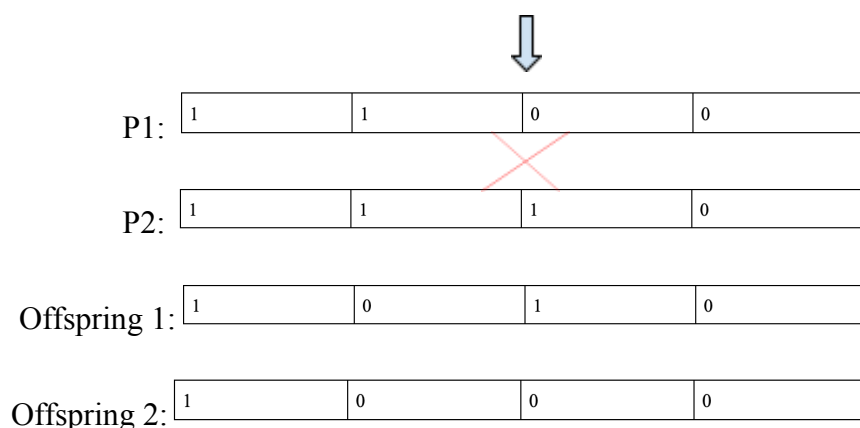


Figure 3: Illustration for crossover between two chromosomes

In figure 3, for example, the locus chosen to be crossed over is gene number. After crossover, the offspring now has the genes of both parent 1 and parent 2.

2.1.3. Mutation

Mutation is a genetic operator that helps to expand the search space, thereby preventing the GAs from being stuck in local optima. The mutation of GA in the knapsack problem is a bit string mutation in which it flips the bit at random positions of the chromosomes. For example, a chromosome with a composition of 1 0 1 0 after mutation at the second bit will be 1 1 1 0.

2.1.4. Premature convergence

Premature convergence causes loss of diversity, which is a problem that many Evolution Computation systems face (Črepinšek et al., 2013). This phenomenon happens when a few fit individuals in the initial population dominate the whole population, preventing the population from exploring potentially better individuals (Andre et al., 2000). Since it is very difficult for the population to move towards a better solution once converged, the population will be stuck in the local optima. Therefore, it is prerequisite that the population needs to be diverse for more exploration to avoid this phenomenon.

2.1.5. Elitism

Elitism is an algorithm that preserves the first best individuals or the few best individuals (the elites) in the generation to the new population (Sharma et al., 2014). This method ensures that good solutions are not lost during the breeding process so that the fitness value of the upcoming generations will increase. In some cases, elitism can improve the performance of the program significantly as it generates a very fit population (Sharma et al., 2014). However, one thing to consider when applying elitism in GA is that elitism makes the algorithm

become much more exploitative. It thus causes premature convergence to happen (Kutubi et al., 2018).

2.1.6. Termination condition

The algorithm is terminated when either one of these conditions is met: two consecutive generations have the same mean fitness value or the limit number of generations is reached.

2.2. Selection strategies

There are several selection strategies such as truncation selection, rank-based selection, deterministic sampling, roulette wheel and tournament selection. Each selection has different characteristics. This paper is investigating the roulette wheel and tournament selection since they are one of the most well-known selection strategies being used for the genetic algorithm.

2.2.1. Roulette Wheel Selection

Roulette wheel selection is the most frequently used selection strategy (Zhong et al., 2006). As suggested by the name, this strategy is influenced by the proportional selecting principle of the physical roulette wheel. The wheel is divided into sections that correspond to the amount of the value of winning, i.e the larger the winning is, the smaller the sector on the spin will be. Therefore, when the wheel is spun, the winning probability will be lower. Similarly, the roulette wheel selection strategy used in the GA also applies this principle but conversely, the higher the fitness, the larger the sector divided on the wheel will be and vice versa.

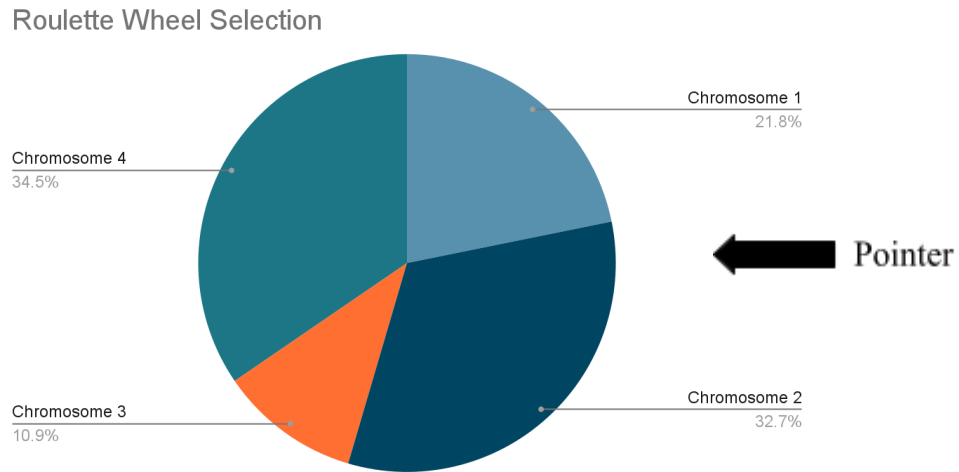


Figure 4: Illustration of the roulette wheel

The probability of an individual i to be selected in the roulette wheel selection can be calculated by the formula below, in which f_i is the fitness of i and l is the number of individuals in the population:

$$p_i = \frac{f_i}{\sum_{n=1}^l f_i}$$

For example, given a population of 4 individuals with the consecutive fitness scores of 10, 20, 30, 40. The probability that the individual number 4 is selected is

$$p_4 = \frac{40}{10+20+30+40} = 0.4$$

Due to its characteristic, roulette wheel selection always gives a chance for all of the individuals in the population, even the weaker ones to be selected. Thus, this trait helps to expand the search space, making this selection more explorative.

2.2.2. Tournament Selection

Tournament selection chooses the individuals merely based on their fitness value. As the name suggests, the algorithm will first randomly choose a certain number of individuals from the population, then it will compare the fitness value between them and finally, choose the one with the highest fitness values to breed and reproduce the next generation. Unlike roulette wheel selection, there is no arithmetical computation based on the fitness value in tournament selection (Zhong et al., 2006). The number of individuals chosen for the tournament is called tournament size.

Although all of the individuals in the population have the same chance to be selected. Since the tournament is merely based on comparing the fitness of the individuals, the individuals with higher fitness value will have a much higher chance to be selected for the next generation, which makes the algorithm become more exploitative.

2.3. The Knapsack Problem

The Knapsack Problem is a typical combinatorial optimization problem with more than 40 years of research (Pan & Zhang, 2018). This problem can be described mathematically as follows: given that you have a knapsack of capacity W , which is the maximum weight that your knapsack can hold. You have a list of n items, each with a weight of w_i and a value of v_i . Our goal is to maximize the value of items that we can bring without exceeding the knapsack capacity (Jaszkiewicz, 2002),

$$\begin{aligned} & \text{maximise } \sum_{i=1}^n v_i \cdot x_i \\ & \text{subject to } \sum_{i=1}^n w_i \cdot x_i \leq W \text{ and } x_i \in \{0; 1\} \end{aligned}$$

x_i is the number of instances of item i to include in the knapsack. The range of x_i can only be either 0 or 1 because the item can only be left behind or taken. Hence, this is why the knapsack problem is also known as the 0/1 knapsack problem.

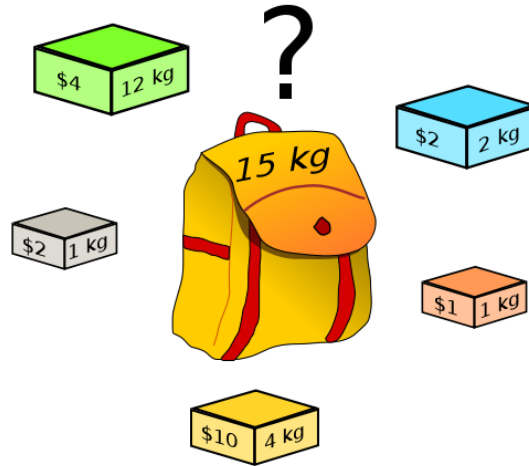


Figure 5: An illustration of the knapsack problem (source: Wikipedia)

The knapsack problem is an NP-C (Non-deterministic Polynomial Completeness) problem. Since there are n items, each with two binary options 0/1, there are 2^n possible combinations of items, making its computation complexity $O(2^n)$ (Kellerer et al., 2004). This problem can be solved by the classical Brute Force algorithm by trying out all possible solutions; however, due to the exponential complexity, this algorithm is only applicable with a small value of n (Pan & Zhang, 2018). That is the reason why other non-deterministic algorithms such as dynamic programming and genetic algorithm are more effective in solving this problem.

1	2	3	4	5
1	0	0	1	1

Figure 6: Binary code for the Knapsack problem with 5 items

Figure 5 exemplifies a solution for the Knapsack problem with 5 items consecutively marked from 1 to 5. The binary option 1 means that the item will be taken and 0 means the item will be left out. So in the example of figure 5, 3 items will be taken: item 1, 4 and 5.

3. Hypothesis

Prior to my research, there has been numerous research about the performance of the genetic algorithm. A research paper done by Jinghui Zhong and the others has found out that tournament selection is more effective in convergence than roulette wheel in solving different functions (Zhong et al., 2006). This case might also be true to my investigation.

Given that tournament selection has the time complexity of $O(n)$, while for roulette wheel is $O(n^2)$ (Sharma et al., 2014), it is certain that the tournament selection is more efficient than the roulette wheel selection in terms of time complexity. Combining this feature with the higher rate of exploitation, I hypothesize that tournament selection will outperform the roulette wheel in terms of convergence rate. However, due to its higher exploitation rate, it is likely that the tournament selection might face premature convergence, especially with elitism. In that case, the fast pace of converging might compensate for the quality of the output solution of the tournament selection. A higher mutation rate and crossover rate might increase the diversity of the population, thus improving the quality of the solution.

Meanwhile, unlike tournament selection, the roulette wheel has a better balance of exploration and exploitation. Since it is more explorative, a diverse search space would lead to a higher chance of finding the best solution compared to that of tournament selection so the output quality of the roulette wheel selection would be better than that of the tournament.

Nevertheless, due to the process of calculating the fitness proportion, this selection will take a longer time to converge and more computational energy than tournament selection.

4. Methodology

In this paper, besides literature research, I also used an empirical approach to compare the two strategies. This section features the detailed experimental procedures and the variables used to determine the results, with reference to the Java code. In order

4.1. The experimental procedures

The GA is run several times with three sets of data and several parameter combinations, i.e different crossover rates, mutation rates and with/without elitism. For each test, the statistics of the GA performance, as well as the graph of the mean fitness by generations are recorded for later analysis.

The detailed procedure is as follow:

- Find and set up the suitable dataset for the Knapsack Problem.
- Set up the program to insert the data fetched from the given dataset including the values and weight of each item in the list, the crossover rate, the mutation rate and the strategy used. Because the original code I used only had a general selection strategy so I went on to write an implementing code for tournament selection and roulette wheel selection.
- Run the program several times using different inputs of the population size.
- Record the total generations the process takes, the fitness score and the generation in which the individual with the best fitness occurs.
- Synthesize the taken data into tables and graphs.

4.2. Independent variables

a. Dataset used

The experiment was conducted using a dataset directory created by Donald Kreher, Douglas Simpson and Silvano Martello, Paolo Toth. The given knapsack has a weight capacity of 750. The item list contains 15 objects with different weights and values. As long as the total weight of items does not exceed the knapsack's capacity, the subset of the objects is considered qualified. The dataset is also given with the most optimal profit and the most optimal selection so that we can compare it with the solutions given by the algorithm to analyze the efficiency of the algorithm (which is further explained in 4.5)

No.	Weight	Value	Optimal selection
1	70	135	1
2	73	139	0
3	77	149	1
4	80	150	0
5	82	156	1
6	87	163	0
7	90	173	1
8	94	184	1
9	98	192	1
10	106	201	0
11	110	210	0
12	113	214	0
13	115	221	0
14	118	229	1
15	120	240	1
Capacity: 750		Optimal profit: 1458	

Figure 7: Dataset for the Knapsack Problem

b. Other parameters

Mutation rate	0.01 and 0.03
Crossover rate	0.85 and 0.95

Population size	100, 200, 300, 400, 500, 750, 1000
Tournament size	5
Elitism	2 chromosomes with the highest fitness value in the population is chosen to be in the next generation.
Maximum generation	5000

4.3. Dependent variables

The dependent variable measured in this experiment is the *number of generations* the algorithm takes to find the most optimal solution, the generation in which the best solution occurs and the fitness of the best individual in the population at the end of the process when the most optimized solution is found or the algorithm reaches its limit population. These variables will then be taken to evaluate the optimization speed (measured by the number of generations including the initial generation) and the optimization reliability (measured by the fitness of the best individual).

4.4. Controlled variables

Variable	Description	Specifications
Computer and operating system used	MacBook Pro with macOS Big Sur	Version: 11.5.2 Processor: 1,4 GHz Quad-Core Intel Core i5 Memory: 8GB 2133 MHz LPDDR3 Serial Number: FVFD240PP3Y1

Integrated Development Environment (IDE) used	The IDE that the program is running on	<p>IDE: IntelliJ IDEA CE 2021.3</p> <p>Build #IC-213.5744.223</p> <p>Runtime version: 11.0.13+7-b1751.19 x86_64</p> <p>Java Runtime Environment:</p> <p>Java Virtual Machine: OpenJDK 64-Bit Server VM by JetBrains s.r.o.</p> <p>macOS 11.5.2</p> <p>GC: G1 Young Generation, G1 Old Generation</p> <p>Memory: 1024M</p> <p>Cores: 8</p>
Algorithm used	The algorithm used in the experiment is in Appendix A	
Functions used	Most of the functions in the program will be the same except from the selecting function	

4.5. Efficiency measure

The efficiency of the performance of two selection strategies is measured based on two factors: convergence rate (speed) and solution reliability (quality). The variables used to

measure both of these factors are given in 4.2 Dependent variables. The convergence speed is measured by the number of generations it takes to find the most optimized solution. The smaller the number of generations is, the more quickly the algorithm converges. Meanwhile, its reliability is measured by the difference between the fitness of the solution found with the most optimized solution that is already known beforehand in the dataset (given in 4.2.a).

5. Experiment Results and Analysis

5.1. Experiment 1

- Crossover rate: 0.85
- Mutation rate: 0.01
- Without elitism

Population size	Roulette Wheel			Tournament		
	No. of generations	Best generation	Best fitness score	No. of generations	Best generation	Best fitness score
100	186	11	1456	33	3	1444
200	707	23	1455	15	5	1445
300	4297	29	1450	46	4	1452
400	1497	10	1452	411	1	1455
500	945	33	1450	572	5	1458
750	5000	20	1456	3365	6	1453
1000	5000	28	1458	5000	6	1458

Since this is the first experiment, not much conclusion can be withdrawn from it. However, we can see that tournament selection converges faster since it takes fewer generations to converge (as shown in the “No. of generations” column) compared to roulette wheel, except for the last run with the population size of 1000. This might be due to the big size of the population but in most other cases, it is certain that the tournament selection is faster in terms

of converging rate. Regarding the fitness score, since GA is non-deterministic, the fitness score varies differently for each run and it is not certain to conclude the difference between the two strategies.

5.2. Experiment 2

- Crossover rate: 0.85
- Mutation rate: 0.01
- With elitism

Population size	Roulette Wheel			Tournament		
	No. of generations	Best generation	Best fitness score	No. of generations	Best generation	Best fitness score
100	8	0	1437	5	0	1440
200	28	16	1446	13	4	1449
300	28	4	1458	9	1	1446
400	26	2	1455	12	0	1439
500	17	0	1449	5	0	1449
750	36	7	1458	5	0	1446
1000	710	2	1458	6	0	1453

Compared to the first experiment, there is a clear difference in both strategies when elitism is added. Starting with roulette wheel, the converging rate has significantly improved as it takes much fewer generations to converge. It seems like without elitism, roulette wheel usually misses the potential best solution as it is not guaranteed to be selected for the next generations. Therefore, it takes a longer time to converge although the best solution is usually found in the early generations. For tournament selection, the best solution is mostly found within the first generation (generation 0) while the fitness score of the solution found is not the best one (we already know the best solution has the fitness value of 1458) regardless of

the population size. Therefore, it is likely that premature has happened to tournament selection as elitism makes the algorithm become much more exploitative.

5.3. Experiment 3

- Crossover rate: 0.95
- Mutation rate: 0.01
- Without elitism

Population size	Roulette Wheel			Tournament		
	No. of generations	Best generation	Best fitness score	No. of generations	Best generation	Best fitness score
100	157	141	1448	13	4	1448
200	199	22	1452	16	7	1458
300	391	16	1458	19	3	1454
400	631	55	1452	70	3	1451
500	5000	12	1458	63	2	1456
750	5000	47	1458	97	3	1454
1000	5000	0	1458	1704	7	1458

For experiments 3 and 4, the crossover rate is increased to see its impact. For roulette wheel selection, the probability that the algorithm found the most optimized solution increased significantly, especially with the big population size. An expansion of the search space might explain this result due to the increased crossover rate, however, the population seems to never converge but rather scatter around as the algorithm only stopped when it reaches the maximum generations. For tournament, similar to the first two experiments, it still takes less time to converge than roulette wheel.

5.4. Experiment 4

- Crossover rate: 0.95
- Mutation rate: 0.01
- With elitism

Population size	Roulette Wheel			Tournament		
	No. of generations	Best generation	Best fitness score	No. of generations	Best generation	Best fitness score
100	12	2	1458	6	0	1442
200	15	1	1451	7	1	1450
300	43	5	1456	13	0	1443
400	22	1	1449	6	0	1444
500	32	1	1451	36	0	1445
750	59	1	1446	16	1	1452
1000	36	2	1448	256	1	1448

When adding in elitism, once again premature occurs in the tournament selection. Hence, the increased crossover rate does not seem to have any influence on tournament in this case. However, this phenomenon surprisingly has also occurred with roulette wheel. Compared to experiment 2 where elitism was also applied, in this experiment, roulette wheel also takes significantly less generation to converge, the only difference is that the probability that it reaches the global maximum is much less than that of experiment 2.

5.5. Experiment 5

- Crossover rate: 0.85
- Mutation rate: 0.03
- Without elitism

Population size	Roulette Wheel			Tournament		
	No. of generations	Best generation	Best fitness score	No. of generations	Best generation	Best fitness score
100	286	100	1449	42	1	1446
200	357	171	1451	445	0	1448
300	5000	5	1458	33	5	1454
400	607	39	1458	1629	3	1458
500	5000	12	1458	1700	7	1458
750	3114	20	1456	3672	3	1458
1000	5000	44	1458	1174	4	1458

In experiment 5 and 6, the mutation rate is increased. This experiment interestingly witnessed a significant increase in the quality of the solutions found in both selection strategies. Especially in tournament selection, from the population size of 400 to 1000, the best fitness score is always found. Despite the similar quality of the solution found, tournament selection still surpasses the roulette wheel in terms of converging rate with a lower number of generations (less run time).

5.6. Experiment 6

- Crossover rate: 0.85
- Mutation rate: 0.03
- With elitism

Population size	Roulette Wheel			Tournament		
	No. of generations	Best generation	Best fitness score	No. of generations	Best generation	Best fitness score
100	8	1	1431	12	2	1440
200	11	0	1446	6	0	1441
300	116	0	1448	99	2	1441
400	156	0	1450	219	0	1446
500	1435	2	1442	685	0	1439
750	1282	0	1446	362	0	1451
1000	663	1	1451	549	0	1451

This experiment has the highest exploitation rate as mutation rate is increased and elitism is applied. Both selection strategies found their best solution in only the first three generations and none of them found the best optimal score. Therefore, it can be concluded that premature has appeared in both selection strategies, making both of them stuck at the local maxima.

6. Conclusion

The experiment has shown that the tournament selection converges much faster as it takes fewer generations to output the solution. It is also more exploitative than roulette wheel selection since the best solutions are usually found in early generations. Moreover, my hypothesis is proven to be correct: due to its high rate of exploitation, premature convergence has occurred when elitism is applied in tournament selection.

On the other hand, experiment 5 shows that an increase in the exploration rate can significantly improve the output of tournament selection. It means that with a balance of exploration and exploitation, tournament selection has a better performance with a fast converging rate and higher quality of solutions found. Meanwhile, the roulette wheel except for experiments 2 and 6 where premature convergence occurred, in most other cases, roulette wheel gives better solutions than tournament. The only problem with the roulette wheel is that it takes a longer time to converge without elitism.

In short, the convergence rate of tournament selection is better than roulette wheel in most cases regardless of the configurations. The quality of the solution is not always assured and is varied with different configurations. However, an increase in mutation rate could fix this problem and improve the performance of the tournament selection. With that being said, tournament selection can be much more effective compared to roulette wheel, specifically with bigger optimization problems.

Bibliography

- Andre, J., Siarry, P., & Dognon, T. (2000, November 30). *An improvement of the standard genetic algorithm fighting premature convergence in continuous optimization*. Advances in Engineering Software. Retrieved March 12, 2022, from <https://www.sciencedirect.com/science/article/abs/pii/S0965997800000703>
- Badiru, K. (1970, January 1). *Knapsack problems; methods, models and applications*. Knapsack Problems; Methods, Models and Applications. Retrieved March 12, 2022, from <https://shareok.org/handle/11244/320340>
- Dasgupta, D., & Michalewicz, Z. (1997). 3.1 Genetic Algorithms. In *Evolutionary algorithms in engineering applications*. essay, Springer-Verlag.
- Hao, J.-K., & Solnon, C. (2019, April 10). *Meta-heuristics and Artificial Intelligence*. HAL Open Science. Retrieved March 12, 2022, from <https://hal.archives-ouvertes.fr/hal-02094881>
- Hristakeva, M., & Shrestha, D. (n.d.). *Solving the 0-1 knapsack problem with Genetic Algorithms*. Retrieved March 12, 2022, from <http://www.sc.ehu.es/ccwbayes/docencia/kzmm/files/AG-knapsack.pdf>
- Hristakeva, M., & Shrestha, D. (n.d.). *Solving the 0-1 knapsack problem with Genetic Algorithms*. Retrieved March 12, 2022, from <http://www.sc.ehu.es/ccwbayes/docencia/kzmm/files/AG-knapsack.pdf>
- Jaskiewicz, A. (2002, November 7). *On the performance of multiple-objective genetic local search on the 0/1 Knapsack Problem - A Comparative Experiment*. IEEE Xplore. Retrieved March 12, 2022, from <https://ieeexplore.ieee.org/document/1027751>

Kaplan, Z. (2018, November 30). *File: Non-Convex Objective Function.gif*. Wikimedia Commons. Retrieved March 10, 2022, from

https://commons.wikimedia.org/wiki/User:OgreBot/Uploads_by_new_users/2018_November_30_00:00

Kellerer, H., Pferschy, U., & Pisinger, D. (2004, January 1). *Introduction to NP-completeness of Knapsack problems: Semantic scholar*. Research Gate. Retrieved March 12, 2022, from

<https://www.semanticscholar.org/paper/Introduction-to-NP-Completeness-of-Knapsack-Kellerer-Pferschy/851cd34a917af34c96874ebd026fad0426fed0c4>

Kreher, D., Simpson, D., Martello, S., & Toth, P. (2014, August 17). *Knapsack_01 data for the 01 knapsack problem*. KNAPSACK_01 - Data for the 01 Knapsack Problem.

Retrieved March 10, 2022, from

https://people.math.sc.edu/Burkardt/datasets/knapsack_01/knapsack_01.html

Kutubi, A. A. R., Hong, M.-G., & Kim, C. (2018, February 28). *Evaluating the Performance of Four Selections in Genetic Algorithms-Based Multispectral Pixel Clustering*. Korea Science. Retrieved March 12, 2022, from

<https://koreascience.or.kr/article/JAKO201807356123567.pdf>

Mayo, M. (2014, June 25). *Knapsack-problem-ga-java/knapsackproblem.java at master · MMMAYO13/Knapsack-problem-ga-java*. GitHub. Retrieved March 12, 2022, from <https://github.com/mmmayo13/knapsack-problem-ga-java/blob/master/knapsack/KnapsackProblem.java>

Mustafa, W., Science, D. of C., R. Plant, S. M. and, N. Zlatareva, M. Rousset, A. L.

and, A. Cheng, B. Z. and, W. Mustafa, M. Kamel, A. L. and, L. Forgy, C., W. Mettrey,

- Al., E., T. Ishida, D. Moldovan, S. K. and, Hassanat, A., Minutolo, A., Arman, N., & Zhong, J. (2003). *Optimization of production systems using genetic algorithms*. International Journal of Computational Intelligence and Applications. Retrieved March 12, 2022, from <https://www.worldscientific.com/doi/abs/10.1142/S1469026803000987>
- Pan, X., & Zhang, T. (2018, August 1). *Comparison and Analysis of Algorithms for the 0/1 Knapsack Problem*. Journal of Physics: Conference Series. Retrieved March 12, 2022, from <https://iopscience.iop.org/article/10.1088/1742-6596/1069/1/012024>
- Razali, N. M., & Geraghty, J. (2011, July 8). *Genetic Algorithm Performance with Different Selection Strategies in Solving TSP*. <http://iaeng.org/>. Retrieved March 12, 2022, from http://www.iaeng.org/publication/WCE2011/WCE2011_pp1134-1139.pdf
- Sharma, P., Wadhwa, A., & Komal. (2014, May). *Analysis of selection schemes for solving an optimization ...* Research Gate. Retrieved March 12, 2022, from https://www.researchgate.net/publication/271156981_Analysis_of_Selection_Schemes_for_Solving_an_Optimization_Problem_in_Genetic_Algorithm
- Yu, X., & Gen, M. (2013). 1.2 What Are Evolutionary Algorithms? In *Introduction to evolutionary algorithms*. essay, Springer London.
- Zhong, J., Hu, X., Zhang, J., & Gu, M. (2006, May 22). *Comparison of performance between different selection strategies on simple genetic algorithms*. IEEE Xplore. Retrieved March 12, 2022, from <https://ieeexplore.ieee.org/abstract/document/1631619>
- Črepinšek, M., Liu, S.-hsi, & Mernik, M. (2013, June). *Exploration and exploitation in evolutionary algorithms: A Survey*. ResearchGate. Retrieved March 12, 2022, from https://www.researchgate.net/publication/243055445_Exploration_and_Exploitation_in_Evolutionary_Algorithms_A_Survey

Appendices

Appendix A - Genetic Algorithm for Knapsack Problem

A.1: KnapsackProblem.java (Matthew Mayo, with modification from the student)

```
/**
 * @filename: KnapsackProblem.java
 * @author: Matthew Mayo
 * @modified: 2014-04-08
 * @description: Creates a KnapsackProblem object based on user input,
 * attempts to solve using a genetic algorithm; outputs
 * algorithm data step-by-step, generates list of optimal
 * items for problem, graphs mean fitness by generation;
 * optional command line argument output_filename will
 * redirect all algorithm details output to output_filename
 * in current directory, will overwrite output_filename
 * contents if file exists
 * @usage: java KnapsackProblem <output_filename>
 */

import java.io.Console;
import java.io.File;
import java.io.FileOutputStream;
import java.io.FileNotFoundException;
import java.io.PrintStream;
import java.lang.StringBuilder;
import java.util.*;

public class KnapsackProblem {

    private boolean verbose = false;
```

```

private boolean mutation = false;
private int selectionOper = 0;
private int crossover_count = 0;
private int clone_count = 0;
private int number_of_items = 0;
private int elitismOper = 0;
private int population_size = 0;
private int maximum_generations = 0;
//private int generation_counter = 1;
private int tournament_size = 0;
private double knapsack_capacity = 0;
private double probab_crossover = 0;
private double probab_mutation = 0;
private double total_fitness_of_generation = 0;
//private ArrayList<Double> tournament = new ArrayList<Double>();
private ArrayList<Double> value_of_items = new ArrayList<Double>();
private ArrayList<Double> weight_of_items = new ArrayList<Double>();
private ArrayList<Double> fitness = new ArrayList<Double>();
private ArrayList<Double> total_weight_of_solution= new
ArrayList<Double>();
private ArrayList<Double> best_fitness_of_generation = new
ArrayList<Double>();
private ArrayList<Double> second_best_fitness_of_generation = new
ArrayList<Double>();
private ArrayList<Double> mean_fitness_of_generation = new
ArrayList<Double>();
private ArrayList<String> population = new ArrayList<String>();
private ArrayList<String> breed_population = new ArrayList<String>();
private ArrayList<String> best_solution_of_generation = new
ArrayList<String>();

```

```

private ArrayList<String> second_best_solution_of_generation = new
ArrayList<String>();

/**
 * Main method
 */
public static void main(String[] args) {

    // Check for command line argument output_filename
    // If filename present, redirect all System.out to file
    try {
        File file_name = new File(args[0]);
        if (file_name.exists()) {
            file_name.delete();
        }
        FileOutputStream fos = new FileOutputStream(file_name, true);
        PrintStream ps = new PrintStream(fos);
        System.setOut(ps);
    } catch (FileNotFoundException e) {
        System.out.println("Problem with output file");
    }

    {
        // Construct KnapsackProblem instance and pass control
        KnapsackProblem knap = new KnapsackProblem();

        // Construct graph of mean fitness by generation
        SimpleGraph graph = new
SimpleGraph(knap.mean_fitness_of_generation,

```

```

        "Mean Fitness by Generation");
    }
}

/**
 * Default constructor
 */
public KnapsackProblem() {

    // Get user input
    this.getInput();

    // Make first generation
    this.buildKnapsackProblem();

    // Output summary
    this.showOptimalList();
}

/**
 * Controls knapsack problem logic and creates first generation
 */
public void buildKnapsackProblem() {

    // Generate initial random population (first generation)
    this.makePopulation();

    // Start printing out summary
    System.out.println("\nInitial Generation:");
    System.out.println("=====");
}

```



```

System.out.println("Population:");

for (int i = 0; i < this.population_size; i++) {
    System.out.println((i + 1) + " - " + this.population.get(i));
}

// Evaluate fitness of initial population members
this.evalPopulation();

// Output fitness summary
System.out.println("\nFitness:");

for (int i = 0; i < this.population_size; i++) {
    System.out.println((i + 1) + " - " + this.fitness.get(i));
}

// Find best solution of generation

this.best_solution_of_generation.add(this.population.get(this.getBestSolution()));

// Find second best solution of generation

this.second_best_solution_of_generation.add(this.population.get(this.getSecondBestSolution()));

// Output best solution of generation
System.out.println("\nBest solution of initial generation: " +
this.best_solution_of_generation.get(0));

// Find mean solution of generation
this.mean_fitness_of_generation.add(this.getMeanFitness());

```

```

        // Output mean solution of generation
        System.out.println("Mean fitness of initial generation: " +
this.mean_fitness_of_generation.get(0));

        // Compute fitness of best solution of generation

this.best_fitness_of_generation.add(this.evalGene(this.population.get(this.ge
tBestSolution())));

        // Compute fitness of second best solution of generation

this.second_best_fitness_of_generation.add(this.evalGene(this.population.get(
this.getSecondBestSolution())));

        // Output best fitness of generation
        System.out.println("Fitness score of best solution of initial
generation: " + this.best_fitness_of_generation.get(0));

        // If maximum_generations > 1, breed further generations
        if (this.maximum_generations > 1) {
            makeFurtherGenerations();
        }
    }

/**
 * Makes further generations beyond first, if necessary
 */
private void makeFurtherGenerations() {

        // Breeding loops maximum_generation number of times at most

```

```

for (int i = 1; i < this.maximum_generations; i++) {

    // Check for stopping criterion
    if ((this.maximum_generations > 3) && (i > 4)) {

        // Previous 2 generational fitness values
        double a = this.mean_fitness_of_generation.get(i - 1);
        double b = this.mean_fitness_of_generation.get(i - 2);
        double c = this.mean_fitness_of_generation.get(i - 3);

        // If both 3 are equal, stop
        if (a == b && b == c) {
            System.out.println("\nStop criterion met");
            maximum_generations = i;
            break;
        }

        /**The threshold is met
        if (this.total_weight_of_solution.get(i) ==
knapsack_capacity && this.fitness.get(i) >
this.best_fitness_of_generation.get(i))
        {
            System.out.println("\nStop criterion met");
            maximum_generations = i;
            break;
        }
        */
    }

    // Reset some counters

```

```

this.crossover_count = 0;

this.clone_count = 0;

this.mutation = false;

// Breed population
if(elitismOper == 0) {
    for(int j = 0; j < this.population_size / 2; j++) {
        this.breedPopulation(i);
    }
} else if(elitismOper == 1) {
    for(int j = 0; j < ((this.population_size / 2) - 1); j++) {
        this.breedPopulation(i);
    }
}

// Clear fitness values of previous generation
this.fitness.clear();

// Evaluate fitness of breed population members
this.evalBreedPopulation();

// Copy breed_population to population
for (int k = 0; k < this.population_size; k++) {
    this.population.set(k, this.breed_population.get(k));
}

// Output population
System.out.println("\nGeneration " + (i + 1) + ":");
if ((i + 1) < 10) {
    System.out.println("=====");
}

```

```

if ((i + 1) >= 10) {
    System.out.println("=====");
}

if ((i + 1) >= 100) {
    System.out.println("=====");
}

System.out.println("Population:");
for (int l = 0; l < this.population_size; l++) {
    System.out.println((l + 1) + " - " + this.population.get(l));
}

// Output fitness summary
System.out.println("\nFitness:");
for (int m = 0; m < this.population_size; m++) {
    System.out.println((m + 1) + " - " + this.fitness.get(m));
}

// Clear breed_population
this.breed_population.clear();

// Find best solution of generation

this.best_solution_of_generation.add(this.population.get(this.getBestSolution
()));

// Find second best solution of generation

this.second_best_solution_of_generation.add(this.population.get(this.getSeco
ndBestSolution()));

// Output best solution of generation

```

```

        System.out.println("\nBest solution of generation " + (i + 1) + ":
" + this.best_solution_of_generation.get(i));

        // Find mean solution of generation
        this.mean_fitness_of_generation.add(this.getMeanFitness());

        // Output mean solution of generation
        System.out.println("Mean fitness of generation: " +
this.mean_fitness_of_generation.get(i));

        // Compute fitness of best solution of generation

this.best_fitness_of_generation.add(this.evalGene(this.population.get(this.ge
tBestSolution())));

        // Compute fitness of second best solution of generation

this.second_best_fitness_of_generation.add(this.evalGene(this.population.get(
this.getSecondBestSolution())));

        // Output best fitness of generation
        System.out.println("Fitness score of best solution of generation "
+ (i + 1) + ": " + this.best_fitness_of_generation.get(i));

        // Output crossover/cloning summary
        System.out.println("Crossover occurred " + this.crossover_count +
" times");
        System.out.println("Cloning occurred " + this.clone_count + "
times");

        if (this.mutation == false) {
            System.out.println("Mutation did not occur");

```

```

    }

    if (this.mutation == true) {
        System.out.println("Mutation did occur");
    }
}

private void stopCriterion() {
    SimpleGraph graph = new SimpleGraph(mean_fitness_of_generation,
        "Mean Fitness by Generation");
}

/**
 * Output KnapsackProblem summary
 */
private void showOptimalList() {

    // Output optimal list of items
    System.out.println("\nOptimal list of items to include in knapsack:
");

    double best_fitness = 0;
    int best_gen = 0;

    // First, find best solution out of generational bests
    for (int z = 0; z < this.maximum_generations - 1; z++) {
        if (this.best_fitness_of_generation.get(z) > best_fitness) {
            best_fitness = this.best_fitness_of_generation.get(z);
            best_gen = z;
        }
    }
}

```

```

    }
}

System.out.println("Best generation is " + best_gen);
System.out.println("Best fitness is " + best_fitness);

// Then, go through that's generation's best solution and output items
String optimal_list = this.best_solution_of_generation.get(best_gen);
for (int y = 0; y < this.number_of_items; y++) {
    if (optimal_list.substring(y, y + 1).equals("1")) {
        System.out.print((y + 1) + " ");
    }
}

System.out.println();

for (int i = 0; i < maximum_generations - 1 ; i++) {
    System.out.println("Generation:" + "\t" + i + "\t" + "Fitness:" +
"\t" + this.best_fitness_of_generation.get(i));
}

}

/**
 * Breeds current population to create a new generation's population
 */
private void breedPopulation(int i) {

    // 2 genes for breeding
    int[] genes;

```



```

        // If population_size is odd #, use elitism to clone best solution of
previous generation

        if (elitismOper == 1) {
            breed_population.add(best_solution_of_generation.get(i-1));

breed_population.add(second_best_solution_of_generation.get(i-1));
        }

        // Increase generation_counter
        //generation_counter = generation_counter + 1;

        // Get positions of pair of genes for breeding
genes = select();

        // Crossover or cloning
crossoverGenes(genes[0], genes[1]);

    }

public int[] select() {
    int[] gene = new int[2];
    if (selectionOper == 1) {
        gene[0] = selectGeneTournament();
        gene[1] = selectGeneTournament();
    } else {
        gene[0] = selectGeneRouletteWheel();
        gene[1] = selectGeneRouletteWheel();
    }
    return gene;
}

```

```

/**
 * Performs mutation, if necessary
 */
private void mutateGene() {

    // Decide if mutation is to be used
    double rand_mutation = Math.random();
    if (rand_mutation <= probab_mutation) {

        // If so, perform mutation
        mutation = true;
        String mut_gene;
        String new_mut_gene;
        Random generator = new Random();
        int mut_point = 0;
        double which_gene = Math.random() * 100;

        // Mutate gene
        if (which_gene <= 50) {
            mut_gene = breed_population.get(breed_population.size() - 1);
            mut_point = generator.nextInt(number_of_items);
            if (mut_gene.substring(mut_point, mut_point + 1).equals("1"))
            {
                new_mut_gene = mut_gene.substring(0, mut_point) + "0" +
mut_gene.substring(mut_point);
                breed_population.set(breed_population.size() - 1,
new_mut_gene);
            }
        }
    }
}

```

```

        if (mut_gene.substring(mut_point, mut_point + 1).equals("0"))
    {
        new_mut_gene = mut_gene.substring(0, mut_point) + "1" +
mut_gene.substring(mut_point);
        breed_population.set(breed_population.size() - 1,
new_mut_gene);
    }
}
if (which_gene > 50) {
    mut_gene = breed_population.get(breed_population.size() - 2);
    mut_point = generator.nextInt(number_of_items);
    if (mut_gene.substring(mut_point, mut_point + 1).equals("1"))
    {
        new_mut_gene = mut_gene.substring(0, mut_point) + "0" +
mut_gene.substring(mut_point);
        breed_population.set(breed_population.size() - 1,
new_mut_gene);
    }
    if (mut_gene.substring(mut_point, mut_point + 1).equals("0"))
    {
        new_mut_gene = mut_gene.substring(0, mut_point) + "1" +
mut_gene.substring(mut_point);
        breed_population.set(breed_population.size() - 2,
new_mut_gene);
    }
}
}
}

/**

```

```

    * Selects a gene for breeding
    *
    * @return int - position of gene in population ArrayList to use for
breeding
    */
private int selectGeneRouletteWheel() {

    // Generate random number between 0 and total_fitness_of_generation
    double rand = Math.random() * total_fitness_of_generation;

    // Use random number to select gene based on fitness level
    for (int i = 0; i < population_size; i++) {
        if (rand <= fitness.get(i)) {
            return i;
        }
        rand = rand - fitness.get(i);
    }

    // Not reachable; default return value
    return 0;
}

/**
    * Tournament selection
    * Written by the student
    * @return
    */

private int selectGeneTournament() {

```

```

//Array of genes selected for the tournament
double[][] tournament = new double[tournament_size][2];

for (int j = 0; j < tournament_size; j++) {
    // Generate random position within the range of 0-population size
    Random r = new Random();
    int rand = r.nextInt(population_size);
    tournament[j][0] = rand;
}

// fill in the tournament array with the position and the fitness
value
for (int i = 0; i < tournament_size; i++) {
    // Select random genes from the population
    tournament[i][1] = fitness.get((int) tournament[i][0]);
}

//Select the best individual
double temp = tournament[0][1];
int gene_position = 0;
for (int n = 0; n < tournament_size; n++) {
    if (temp <= tournament[n][1]) {
        temp = tournament[n][1];
        gene_position = (int) tournament[n][0];
    }
}

return gene_position;
}

```

```

/**
 * Performs either crossover or cloning
 */
private void crossoverGenes(int gene_1, int gene_2) {

    // Strings to hold new genes
    String new_gene_1;
    String new_gene_2;

    // Decide if crossover is to be used
    double rand_crossover = Math.random();
    if (rand_crossover <= probab_crossover) {
        // Perform crossover
        crossover_count = crossover_count + 1;
        Random generator = new Random();
        int cross_point = generator.nextInt(number_of_items) + 1;

        // Cross genes at random spot in strings
        new_gene_1 = population.get(gene_1).substring(0, cross_point) +
population.get(gene_2).substring(cross_point);
        new_gene_2 = population.get(gene_2).substring(0, cross_point) +
population.get(gene_1).substring(cross_point);

        // Add new genes to breed_population
        breed_population.add(new_gene_1);
        breed_population.add(new_gene_2);
    } else {
        // Otherwise, perform cloning
        clone_count = clone_count + 1;
        breed_population.add(population.get(gene_1));
        breed_population.add(population.get(gene_2));
    }
}

```

```

    }

    // Check if mutation is to be performed
    mutateGene();
}

/**
 * Gets best solution in population
 *
 * @return int - position of best solution in population
 */
private int getBestSolution() {
    int best_position = 0;
    double this_fitness = 0;
    double best_fitness = 0;
    for (int i = 0; i < population_size; i++) {
        this_fitness = evalGene(population.get(i));
        if (this_fitness > best_fitness) {
            best_fitness = this_fitness;
            best_position = i;
        }
    }
    return best_position;
}

/**
 * Gets second best solution in population
 *
 * @return int - position of second best solution in population
 */

```

```

private int getSecondBestSolution() {

    int second_best_position = 0;

    int best_position = 0;

    double this_fitness = 0;

    double best_fitness = evalGene(population.get(0));

    double second_best_fitness = evalGene(population.get(0));

    for(int i = 1; i < population_size;i++)

    {

        this_fitness = evalGene(population.get(i));

        if (this_fitness > best_fitness) {

            second_best_fitness = best_fitness;

            second_best_position = best_position;

            best_fitness = this_fitness;

            best_position = i;

        }

        if (this_fitness > second_best_fitness && this_fitness !=
best_fitness) {

            second_best_fitness = this_fitness;

            second_best_position = i;

        }

    }

    return second_best_position;

}

/**
 * Gets mean fitness of generation
 */

private double getMeanFitness() {

    double total_fitness = 0;

```



```

double mean_fitness = 0;
for (int i = 0; i < population_size; i++) {
    total_fitness = total_fitness + fitness.get(i);
}
mean_fitness = total_fitness / population_size;
return mean_fitness;
}

/**
 * Evaluates entire population's fitness, by filling fitness ArrayList
 * with fitness value of each corresponding population member gene
 */
private void evalPopulation() {
    total_fitness_of_generation = 0;
    for (int i = 0; i < population_size; i++) {
        double temp_fitness = evalGene(population.get(i));
        fitness.add(temp_fitness);
        total_fitness_of_generation = total_fitness_of_generation +
temp_fitness;
    }
}

/**
 * Evaluates entire breed_population's fitness, by filling breed_fitness
ArrayList
 * with fitness value of each corresponding breed_population member gene
 */
private void evalBreedPopulation() {
    total_fitness_of_generation = 0;

```

```

    for (int i = 0; i < population_size; i++) {
        double temp_fitness = evalGene(breed_population.get(i));
        fitness.add(temp_fitness);
        total_fitness_of_generation = total_fitness_of_generation +
temp_fitness;
    }
}

/**
 * Evaluates a single gene's fitness, by calculating the total_weight
 * of items selected by the gene
 *
 * @return double - gene's total fitness value
 */
private double evalGene(String gene) {
    double total_weight = 0;
    double total_value = 0;
    double fitness_value = 0;
    double difference = 0;
    char c = '0';

    // Get total_weight associated with items selected by this gene
    for (int j = 0; j < number_of_items; j++) {
        c = gene.charAt(j);
        // If chromosome is a '1', add corresponding item position's
        // weight to total weight
        if (c == '1') {
            total_weight = total_weight + weight_of_items.get(j);
            total_value = total_value + value_of_items.get(j);
        }
    }
}

```

```

    }

    // Check if gene's total weight is less than knapsack capacity
    difference = knapsack_capacity - total_weight;
    if (difference >= 0) {
        // This is acceptable; calculate a fitness_value
        // Otherwise, fitness_value remains 0 (default), since knapsack
        // cannot hold all items selected by gene
        // Fitness value is simply total value of acceptable permutation,
        // and for unacceptable permutation is set to '0'
        fitness_value = total_value;
        this.total_weight_of_solution.add(total_weight);
    }

    // Return fitness value
    return fitness_value;
}

/**
 * Makes a population by filling population ArrayList with strings of
 * length number_of_items, each element a gene of randomly generated
 * chromosomes (1s and 0s)
 */
private void makePopulation() {
    for (int i = 0; i < population_size; i++) {
        // Calls makeGene() once for each element position
        population.add(makeGene());
    }
}
}

```

```

/**
 * Generates a single gene, a random String of 1s and 0s
 *
 * @return String - a randomly generated gene
 */
private String makeGene() {

    // StringBuilder builds gene, one chromosome (1 or 0) at a time
    StringBuilder gene = new StringBuilder(number_of_items);

    // Each chromosome
    char c;

    // Loop creating gene
    for (int i = 0; i < number_of_items; i++) {
        c = '0';
        double rnd = Math.random();
        // If random number is greater than 0.5, chromosome is '1'
        // If random number is less than 0.5, chromosome is '0'
        if (rnd > 0.5) {
            c = '1';
        }
        // Append chromosome to gene
        gene.append(c);
    }
    // StringBuilder object to string; return
    return gene.toString();
}

/**

```

```

    * Collects user to be used as parameters for knapsack problem
    */
private void getInput() {

    try {

        File myObj = new File("dataset.txt");
        Scanner s = new Scanner(myObj);
        while (s.hasNext()) {

            /**
             * // Population size
             *
             *          population_size = s.nextInt();
             */

            //Selection Strategy
            selectionOper = s.nextInt();

            //Elitism
            elitismOper = s.nextInt();

            // Tournament size
            tournament_size = s.nextInt();

            // Maximum number of generations
            maximum_generations = s.nextInt();

            // Crossover probability
            probab_crossover = s.nextDouble();

            // Mutation rate
            probab_mutation = s.nextDouble();

```

```

// Number of items
number_of_items = s.nextInt();

// Value of each item
for (int i = 0; i < number_of_items; i++) {
    value_of_items.add(s.nextDouble());
}

// Weight of each item
for (int i = 0; i < number_of_items; i++) {
    weight_of_items.add(s.nextDouble());
}

// Capacity of knapsack
knapsack_capacity = s.nextInt();
}
s.close();

// Hold user input, line by line
String input;

// Initialize console for user input
Console c = System.console();
if (c == null) {
    System.err.println("No console.");
    System.exit(1);
}

// Population size
input = c.readLine("Enter the population size: ");
if (isInteger(input)) {

```

```

        population_size = Integer.parseInt(input);
    } else {
        System.out.println("Not a number. Please try again.");
        System.exit(1);
    }

} catch (FileNotFoundException e) {
    System.out.println("An error occurred.");
    e.printStackTrace();
}
}

```

```

/**
 * Determines if input string can be converted to integer
 *
 * @param String - string to be checked
 * @return boolean - whether or not string can be converted
 */
public static boolean isInteger(String str) {
    try {
        Integer.parseInt(str);
    } catch (NumberFormatException e) {
        return false;
    }
    return true;
}

```

```

/**

```

```

    * Determines if input string can be converted to double
    *
    * @param String - string to be checked
    * @return boolean - whether or not string can be converted
    */
public static boolean isDouble(String str) {
    try {
        Double.parseDouble(str);
    } catch (NumberFormatException e) {
        return false;
    }
    return true;
}

} // KnapsackProblem

```

Roulette Wheel Selection (written by the student)

```

private int selectGeneRouletteWheel() {
    // Generate random number between 0 and total_fitness_of_generation
    double rand = Math.random() * total_fitness_of_generation;

    // Use random number to select gene based on fitness level
    for (int i = 0; i < population_size; i++) {
        if (rand <= fitness.get(i)) {
            return i;
        }
        rand = rand - fitness.get(i);
    }
    // Not reachable; default return value
    return 0;
}

```



```
}
```

Tournament Selection (written by the student)

```
private int selectGeneTournament() {  
    //Array of genes selected for the tournament  
    double[][] tournament = new double[tournament_size][2];  
  
    for (int j = 0; j < tournament_size; j++) {  
        // Generate random position within the range of 0-population size  
        Random r = new Random();  
        int rand = r.nextInt(population_size);  
        tournament[j][0] = rand;  
    }  
  
    // fill in the tournament array with the position and the fitness value  
    for (int i = 0; i < tournament_size; i++) {  
        // Select random genes from the population  
        tournament[i][1] = fitness.get((int) tournament[i][0]);  
    }  
  
    //Select the best individual  
    double temp = tournament[0][1];  
    int gene_position = 0;  
    for (int n = 0; n < tournament_size; n++) {  
        if (temp <= tournament[n][1]) {  
            temp = tournament[n][1];  
            gene_position = (int) tournament[n][0];  
        }  
    }  
  
    return gene_position;  
}
```

A.2: SimpleGraph.java

```
/**
 * @filename:      SimpleGraph.java
 * @author:        Matthew Mayo
 * @modified:      2014-04-08
 * @description:   Creates a SimpleGraph object based on supplied ArrayList
 *                of data points; draws graph, adds points, lines,
appropriate
 *                hatch marks; must supply ArrayList of data points to
plot
 *                and title of graph to display
 * @usage:         java SimpleGraph <data_points> <graph_title>
 * @note:          Inspiration for, and adapted code, comes from:
 *
http://stackoverflow.com/questions/8693342/drawing-a-simple-line-graph-in-j
ava
 */

import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.FontMetrics;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Point;
import java.awt.RenderingHints;
import java.awt.Stroke;
import java.util.ArrayList;
import javax.swing.JFrame;
import javax.swing.JPanel;
```

```

public class SimpleGraph extends JPanel {

    private int width = 800;
    private int height = 400;
    private int padding = 25;
    private int label_padding = 25;
    private int point_width = 6;
    private int number_y_divisions = 0;
    private Color line_color = new Color(44, 102, 230, 180);
    private Color point_color = Color.BLACK;
    private Color grid_color = new Color(200, 200, 200, 200);
    private static final Stroke GRAPH_STROKE = new BasicStroke(2f);
    private String graph_title = "";
    private ArrayList<Double> data_points;

    /**
     * Main method (for testing directly from this class)
     */
    public static void main(String[] args) {

        // Create an ArrayList<Double> of data_points
        ArrayList<Double> test_data = new ArrayList<Double>();

        // Add points to data_points
        test_data.add(1.0);
        test_data.add(9.2);
        test_data.add(5.7);
        test_data.add(7.9);
    }
}

```

```

test_data.add(2.4);
test_data.add(11.5);

// Set a graph title
String test_title = "Graph title goes here";

// Pass data_points and graph_title to SimpleGraph constructor
SimpleGraph test = new SimpleGraph(test_data, test_title);
}

/**
 * Default constructor
 */
public SimpleGraph(ArrayList<Double> data_points, String graph_title) {

    // Set data points data set and graph title
    this.data_points = data_points;
    this.graph_title = graph_title;

    // Set number of y divisions by finding difference between
    // max and min data points
    number_y_divisions = getMaxDataPoint() - getMinDataPoint();

    // Set preferred size of panel
    this.setPreferredSize(new Dimension(800, 600));

    // Create content frame, add to panel
    JFrame frame = new JFrame(graph_title);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}

```

```

    frame.getContentPane().add(this);

    frame.pack();

    frame.setLocationRelativeTo(null);

    frame.setVisible(true);

}

/**
 * Creates and draws graph to specification
 * @param Graphics - What to be drawn
 */
@Override
public void paintComponent(Graphics g) {

    super.paintComponent(g);

    Graphics2D g2 = (Graphics2D)g;

    g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
        RenderingHints.VALUE_ANTIALIAS_ON);

// Set scales

    double xScale = ((double) getWidth() - (2 * padding) - label_padding)
        / (data_points.size() - 1);

    double yScale = ((double) getHeight() - 2 * padding - label_padding)
        / (getMaxDataPoint() - getMinDataPoint());

// Create array of Point objects from passed in array of Doubles
    ArrayList<Point> graphPoints = new ArrayList<>();

    for (int i = 0; i < data_points.size(); i++) {

        int x1 = (int) (i * xScale + padding + label_padding);

        int y1 = (int) ((getMaxDataPoint() - data_points.get(i)) * yScale

```

```

        + padding);

        graphPoints.add(new Point(x1, y1));
    }

    // Draw white background
    g2.setColor(Color.WHITE);

    g2.fillRect(padding + label_padding, padding, getWidth() - (2 *
padding)

        - label_padding, getHeight() - 2 * padding - label_padding);
    g2.setColor(Color.BLACK);

    // Create hatch marks and grid lines for y axis
    for (int i = 0; i < number_y_divisions + 1; i++) {
        if(number_y_divisions == 0) {
number_y_divisions = number_y_divisions + 1;
        }

        int x0 = padding + label_padding;

        int x1 = point_width + padding + label_padding;

        int y0 = getHeight() - ((i * (getHeight() - padding * 2
        - label_padding)) / number_y_divisions + padding +
label_padding);

        int y1 = y0;

        if (data_points.size() > 0) {
            g2.setColor(grid_color);

            g2.drawLine(padding + label_padding + 1 + point_width, y0,
                getWidth() - padding, y1);

            g2.setColor(Color.BLACK);

            String yLabel = ((int) (getMinDataPoint() + (getMaxDataPoint()
- getMinDataPoint()) *
                ((i * 1.0) / number_y_divisions))) + " ";

            FontMetrics metrics = g2.getFontMetrics();

```

```

        int labelWidth = metrics.stringWidth(yLabel);
        g2.drawString(yLabel, x0 - labelWidth - 5, y0
            + (metrics.getHeight() / 2) - 3);
    }
    g2.drawLine(x0, y0, x1, y1);
}

// Create hatch marks and grid lines for x axis
for (int i = 0; i < data_points.size(); i++) {
    if (data_points.size() > 1) {
        int x0 = i * (getWidth() - padding * 2 - label_padding)
            / (data_points.size() - 1) + padding + label_padding;
        int x1 = x0;
        int y0 = getHeight() - padding - label_padding;
        int y1 = y0 - point_width;
        if ((i % ((int) ((data_points.size() / 20.0)) + 1)) == 0) {
            g2.setColor(grid_color);
            g2.drawLine(x0, getHeight() - padding - label_padding - 1
                - point_width, x1, padding);
            g2.setColor(Color.BLACK);
            String xLabel = (i + 1) + "";
            FontMetrics metrics = g2.getFontMetrics();
            int labelWidth = metrics.stringWidth(xLabel);
            g2.drawString(xLabel, x0 - labelWidth / 2, y0
                + metrics.getHeight() + 3);
        }
        g2.drawLine(x0, y0, x1, y1);
    }
}

// Create x and y axes

```

```

        g2.drawLine(padding + label_padding, getHeight() - padding -
label_padding,
                padding + label_padding, padding);
        g2.drawLine(padding + label_padding, getHeight() - padding -
label_padding,
                getWidth() - padding, getHeight() - padding - label_padding);

// Draw lines
Stroke oldStroke = g2.getStroke();
g2.setColor(line_color);
g2.setStroke(GRAPH_STROKE);
for (int i = 0; i < graphPoints.size() - 1; i++) {
    int x1 = graphPoints.get(i).x;
    int y1 = graphPoints.get(i).y;
    int x2 = graphPoints.get(i + 1).x;
    int y2 = graphPoints.get(i + 1).y;
    g2.drawLine(x1, y1, x2, y2);
}

// Draw points
g2.setStroke(oldStroke);
g2.setColor(point_color);
for (int i = 0; i < graphPoints.size(); i++) {
    int x = graphPoints.get(i).x - point_width / 2;
    int y = graphPoints.get(i).y - point_width / 2;
    int ovalW = point_width;
    int ovalH = point_width;
    g2.fillOval(x, y, ovalW, ovalH);
}
}

```



```

/**
 * Returns minimum data point in data_points set
 * @return int - Minimum data point in set
 */
private int getMinDataPoint() {
    int min_data_point = Integer.MAX_VALUE;
    Integer dp_conv = 0;
    for (Double data_point : data_points) {
        dp_conv = (int) data_point.doubleValue();
        min_data_point = Math.min(min_data_point, dp_conv);
    }
    return min_data_point;
}

```

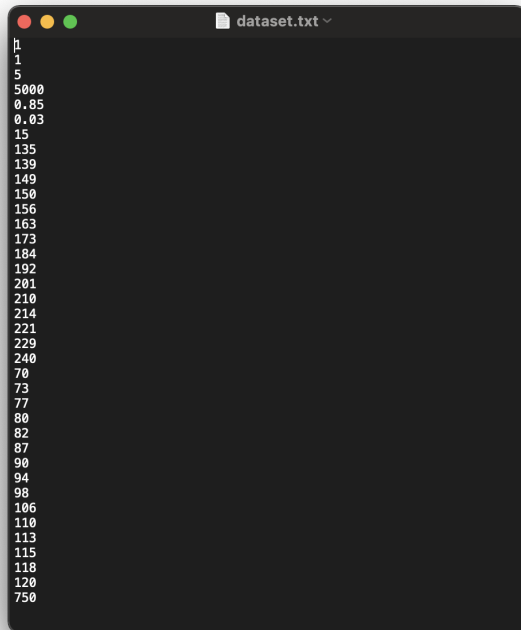
```

/**
 * Returns maximum data point in data_points set
 * @return int - Maximum data point in set
 */
private int getMaxDataPoint() {
    int max_data_point = Integer.MIN_VALUE;
    Integer dp_conv = 0;
    for (Double data_point : data_points) {
        dp_conv = (int) data_point.doubleValue() + 1;
        max_data_point = Math.max(max_data_point, dp_conv);
    }
    return max_data_point;
}

```

```
} // SimpleGraph
```

Appendix B - Sample of Input text



```
dataset.txt
1
5
5000
0.85
0.03
15
135
139
149
150
156
163
173
184
192
201
210
214
221
229
240
70
73
77
80
82
87
90
94
98
106
110
113
115
118
120
750
```

Appendix C - License to use the code

<https://github.com/mmmayo13/knapsack-problem-ga-java/blob/master/LICENSE>

The MIT License
(MIT)

Copyright (c) 2014 Matthew Mayo

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is

furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.