# Evolutionary Algorithm Lab Report

Yiping Huang, Tianchen Luo
*Department of Data Science and Artificial Intelligence*
*Maastricht University*
Maastricht, The Netherlands

***Abstract*—The document reports the test results of the EA(Evolutionary Algorithm) on different instances of problems, i.e., the 0-1 knapsack problems and the TSP(Traveling Salesman Problem).**

***Index Terms*—EA, 0-1 knapsack problem, TSP problem, Combinatorial Optimization**

## I. 0-1 KNAPSACK PROBLEM

### *A. Encoding Scheme*

For 0-1 knapsack problem, the binary code genotype is used, since it is the most natural representation of the solution, i.e., the 0 and 1 represents the absence and presence of an item respectively.

### *B. Repair Function*

For combinatorial optimization problem, one of the extra things we need to consider is how to deal with constraint. For this problem, we need to limit the total weight of included items below to a certain upper bound.

There are different techniques to deal with this, such as introducing a punishment term in the objective function, or introducing a repair step to make the solution feasible again.

Both approaches have pros and cons. The punishment approach is efficient for computing, by simply including an extra term to the fitness function, e.g., linear, log or quadratic that is proportional to overflow due to the capacity constraint [MA94]. Repair function, while safely perserving the feasibility of solutions, limits the search space however. However, results showed that for the hard constraint problem such as 0-1 knapsack problem, the repair approach should be adopted since the punishment approach produces constantly infeasble solutions [MA94].

*1) Random Repair:* This is the most simple repair function. Randomly choose a loci with $1$ value in it, and then replace it with $0$ value until the solution is feasible again.

*2) Greedy Repair:* The greedy approach uses heuristics that the item with a smaller profit to weight ratio should be removed first in the repair process, as can be easily understood.

### *C. Selection*

Again, there are numerous approaches. We used here the RWS (Roulette Wheel Selection), which samples item with a discrete probability distribution that reflects the fitness of each item. In implementation, we found that the numerical instability caused the total probability of all items not sum up to 1, which causes the 'numpy.choice' function not able to do the sampling for us. We implemented the sampling thus from scratch. Specifically, we used Gibbs distribution with temperature T hyperparameter. When setting $T = 1$, this is equal to softmax function as commonly used in machine learning. We also used the property that $softmax(\mathbf{x}+c) = softmax(\mathbf{x})$ to shift the mean value by the max of the fitness value to avoid numerical overflow and underflow.

### *D. Crossover*

Crossover in the case of knapsack can have bigger destructive effect, so we only used single-point crossover. This way, we can maintain the balance between the exploration and exploitation.

### *E. Mutation*

Nothing special here, just random mutation with small probability.

### *F. Dataset*

We used dataset produced by the generator [Pis99] to test the algorithm instead of generating artificial dataset by our own, as the former dataset includes a few criteria which are deemed important to evaluate the algorithm, such as the correlation between the profit and weight, e.g., strong, weak or uncorellated, the size of the items, the sparcity of the solution vector etc. We didn't use the generator to generate the dataset, but used the dataset that is available in the public domain, which can be found at: http://artemisa.unicauca.edu.co/~johnyortega/instances_01_KP/. More specifically, the first digit in the filename corresponds to the correlation level between the weight and the fitness value, where $1, 2, 3$ corresponds to uncorellated, weakly correlated and strongly correlated data respectively. The second number separated by the dash is the number of items in the data.

### *G. Result*

Here we show the result for the experiment. Table I shows the deviation from the optimal values in percentage with population size of 10. Table II shows the running time for a total of 100 generations in the time column. The found column indicates when the best solution was found in terms of the number of generations. The weight found by the GA is also listed together with the weight found by the optimal solution for comparison. Figure 1 shows the typical change of average fitness over the run of generations, using the input from the file knapPI_1_2000_1000_1. We see that increase in fitness happens mostly in the early stage, which is general the case for most GA algorithms, where GA tries to explore as much as it can in the early stage and then use local search in the later stage to narrow down the result.

| file | ga_sol | actual_sol | deviation |
|---|---|---|---|
| knapPI_1_5000_1000_1 | 221927 | 276457 | 0.20 |
| knapPI_2_100_1000_1 | 1512 | 1514 | 0.00 |
| knapPI_2_2000_1000_1 | 17022 | 18051 | 0.06 |
| knapPI_3_200_1000_1 | 2693 | 2697 | 0.00 |
| knapPI_1_500_1000_1 | 27553 | 28857 | 0.05 |
| knapPI_1_200_1000_1 | 11238 | 11238 | 0.00 |
| knapPI_1_10000_1000_1 | 440860 | 563647 | 0.22 |
| knapPI_1_2000_1000_1 | 94002 | 110625 | 0.15 |
| knapPI_3_1000_1000_1 | 13376 | 14390 | 0.07 |
| knapPI_2_5000_1000_1 | 41280 | 44356 | 0.07 |
| knapPI_3_500_1000_1 | 6914 | 7117 | 0.03 |
| knapPI_2_10000_1000_1 | 81867 | 90204 | 0.09 |
| knapPI_3_2000_1000_1 | 26016 | 28919 | 0.10 |
| knapPI_3_100_1000_1 | 2390 | 2397 | 0.00 |
| knapPI_1_1000_1000_1 | 49987 | 54503 | 0.08 |
| knapPI_3_10000_1000_1 | 124091 | 146919 | 0.16 |
| knapPI_2_200_1000_1 | 1634 | 1634 | 0.00 |
| knapPI_3_5000_1000_1 | 62798 | 72505 | 0.13 |
| knapPI_2_1000_1000_1 | 8761 | 9052 | 0.03 |
| knapPI_2_500_1000_1 | 4495 | 4566 | 0.02 |
| knapPI_1_100_1000_1 | 8929 | 9147 | 0.02 |

TABLE I: Deviation with population size =10

| file | found | ga_weight | actual_weight | time |
|---|---|---|---|---|
| knapPI_1_5000_1000_1 | 76 | 24849 | 25016 | 9.05 |
| knapPI_2_100_1000_1 | 24 | 953 | 991 | 0.27 |
| knapPI_2_2000_1000_1 | 78 | 10009 | 10010 | 4.42 |
| knapPI_3_200_1000_1 | 84 | 993 | 997 | 0.45 |
| knapPI_1_500_1000_1 | 85 | 2498 | 2543 | 1.11 |
| knapPI_1_200_1000_1 | 4 | 987 | 987 | 0.48 |
| knapPI_1_10000_1000_1 | 63 | 49714 | 49877 | 19.99 |
| knapPI_1_2000_1000_1 | 87 | 10003 | 10011 | 3.93 |
| knapPI_3_1000_1000_1 | 43 | 4976 | 4990 | 2.04 |
| knapPI_2_5000_1000_1 | 99 | 24999 | 25016 | 10.28 |
| knapPI_3_500_1000_1 | 74 | 2514 | 2517 | 1.14 |
| knapPI_2_10000_1000_1 | 76 | 49703 | 49877 | 19.91 |
| knapPI_3_2000_1000_1 | 73 | 9816 | 9819 | 3.90 |
| knapPI_3_100_1000_1 | 22 | 990 | 997 | 0.24 |
| knapPI_1_1000_1000_1 | 78 | 4965 | 5002 | 1.98 |
| knapPI_3_10000_1000_1 | 44 | 49491 | 49519 | 19.76 |
| knapPI_2_200_1000_1 | 67 | 1006 | 1006 | 0.45 |
| knapPI_3_5000_1000_1 | 64 | 24798 | 24805 | 9.71 |
| knapPI_2_1000_1000_1 | 98 | 4993 | 5002 | 2.17 |
| knapPI_2_500_1000_1 | 55 | 2539 | 2543 | 1.03 |
| knapPI_1_100_1000_1 | 3 | 972 | 985 | 0.24 |

TABLE II: running time(s) for 100 generations with population size 10

Similarly, Table III shows the deviation with population of size 100, and table IV shows the running time, weight, and the number of generations passed to find the best solution as well. Due to the enlarged population size, the result we got is closer to the actual solution compared with setting previously. The running time also increases 10 times in proportion with size.

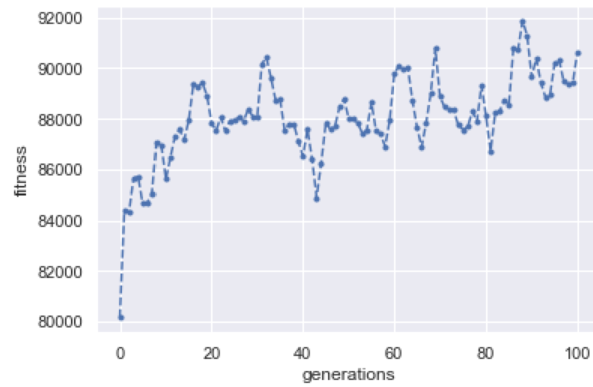



Fig. 1: Average Fitness Change (population size = 10) with File knapPI_1_2000_1000_1 as Input

| file | ga_sol | actual_sol | deviation |
|---|---|---|---|
| knapPI_1_5000_1000_1 | 234227 | 276457 | 0.15 |
| knapPI_2_100_1000_1 | 1512 | 1514 | 0.00 |
| knapPI_2_2000_1000_1 | 17506 | 18051 | 0.03 |
| knapPI_3_200_1000_1 | 2696 | 2697 | 0.00 |
| knapPI_1_500_1000_1 | 28834 | 28857 | 0.00 |
| knapPI_1_200_1000_1 | 11238 | 11238 | 0.00 |
| knapPI_1_10000_1000_1 | 456254 | 563647 | 0.19 |
| knapPI_1_2000_1000_1 | 102992 | 110625 | 0.07 |
| knapPI_3_1000_1000_1 | 13971 | 14390 | 0.03 |
| knapPI_2_5000_1000_1 | 42101 | 44356 | 0.05 |
| knapPI_3_500_1000_1 | 7016 | 7117 | 0.01 |
| knapPI_2_10000_1000_1 | 83179 | 90204 | 0.08 |
| knapPI_3_2000_1000_1 | 26818 | 28919 | 0.07 |
| knapPI_3_100_1000_1 | 2396 | 2397 | 0.00 |
| knapPI_1_1000_1000_1 | 52614 | 54503 | 0.03 |
| knapPI_3_10000_1000_1 | 127090 | 146919 | 0.13 |
| knapPI_2_200_1000_1 | 1634 | 1634 | 0.00 |
| knapPI_3_5000_1000_1 | 64392 | 72505 | 0.11 |
| knapPI_2_1000_1000_1 | 8928 | 9052 | 0.01 |
| knapPI_2_500_1000_1 | 4557 | 4566 | 0.00 |
| knapPI_1_100_1000_1 | 9147 | 9147 | 0.00 |

TABLE III: Deviation with population size =100

## II. TSP Problem

### A. Encoding Scheme

For TSP problem, various encoding schemes exist. The most natural and commonly used scheme is to use the order of traversal of nodes as genotype, i.e., the path code. The search domain is $n!$. Thus it contains $2n$ redundancy, since every instance of phenotype corresponds to $2n$ genotypes, each starts from a different node, and either traverse in order or in reverse order. However, many operators all based on this coding scheme. This is also the encoding scheme we choose to use in the experiment.

### B. Fitness Function

Unlike previous problem, we have here a minimization problem. To generate positive fitness value which can be used in the selection mechanism, we do the simple transform of distance: first we get the maximum of distance among population, and then we use this value minus the distance to get the fitness value for each individual.

| file | found | ga_weight | actual_weight | time |
|---|---|---|---|---|
| knapPI_1_5000_1000_1 | 91 | 24895 | 25016 | 93.59 |
| knapPI_2_100_1000_1 | 4 | 953 | 991 | 2.15 |
| knapPI_2_2000_1000_1 | 83 | 9994 | 10010 | 39.79 |
| knapPI_3_200_1000_1 | 2 | 996 | 997 | 4.36 |
| knapPI_1_500_1000_1 | 28 | 2528 | 2543 | 10.50 |
| knapPI_1_200_1000_1 | 9 | 987 | 987 | 4.21 |
| knapPI_1_10000_1000_1 | 37 | 49726 | 49877 | 202.95 |
| knapPI_1_2000_1000_1 | 77 | 9998 | 10011 | 39.80 |
| knapPI_3_1000_1000_1 | 75 | 4971 | 4990 | 19.46 |
| knapPI_2_5000_1000_1 | 33 | 24999 | 25016 | 99.27 |
| knapPI_3_500_1000_1 | 14 | 2516 | 2517 | 9.80 |
| knapPI_2_10000_1000_1 | 28 | 49875 | 49877 | 201.76 |
| knapPI_3_2000_1000_1 | 97 | 9818 | 9819 | 39.18 |
| knapPI_3_100_1000_1 | 10 | 996 | 997 | 2.27 |
| knapPI_1_1000_1000_1 | 69 | 4996 | 5002 | 20.34 |
| knapPI_3_10000_1000_1 | 69 | 49390 | 49519 | 220.02 |
| knapPI_2_200_1000_1 | 2 | 1006 | 1006 | 4.26 |
| knapPI_3_5000_1000_1 | 17 | 24792 | 24805 | 95.54 |
| knapPI_2_1000_1000_1 | 28 | 4995 | 5002 | 19.82 |
| knapPI_2_500_1000_1 | 37 | 2539 | 2543 | 9.82 |
| knapPI_1_100_1000_1 | 1 | 985 | 985 | 2.11 |

TABLE IV: running time(s) for 100 generations with population size 100

### C. Initialzation

To construct an initial population. We could simply permutate the node order, and each permutation is a valid solution (we assume the graph is fully connected, otherwise we can always add dummy edges and setting large values on these edges). Since the search space is huge (n!), we could guide the search through generation of good candidates at the start. Many heuristics exist again, the most simple one being the greedy method which simply starts from a random chosen node and find the next closest neighbor not visited yet and iterate until all nodes have been traversed. This is the approach we used in the experiment.

### D. Selection

For the experiment, we tried both RWS method and tournament selection method. For the tournament selection method, we used the unbiased version, which uses permutation method to choose the candidates for the tournament. We simply permutate the population $k$ times, $k$ being the size of the tournament. This way, each node participate exactly $k$ times in the tournament, so as to avoid the unlucky situation where the fitter individual is not showing up the tournament.

### E. Mutation

If we used single point mutation, then we lose the feasibility of the solution, i.e., we traverse some nodes twice. Here we used exchanged mutation operator here as suggested by [Ban90], which simply exchange two node positions at the same time.

### F. Crossover

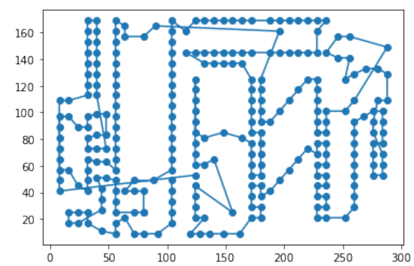
We used PMX(Partially Mapped Crossover) operator as suggested by [GL+85]. The method first exchange the subroute of two parents to form two children, then fill in the rest positions in children sequentially, using the corresponding parent's node first. If the parent's node is in conflict with nodes in the subroute, we resolve the conflict by setting up two maps in the process of exchanging subroutes. Each gives a mapping between two subroutes in different direction. Then we can simply resolve the conflict by looking through the map until the map points to an available node.

### G. Dataset

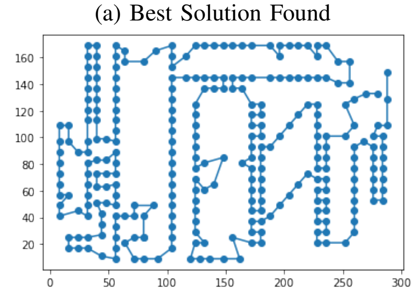
TSPLIB is the most known library for TSP datasets. Here we used a280 from the library. It comes from the real problem of finding the best route for machine to drill holes.

### H. Result

Figure 2a shows the best solution found by our algorithm with minimum distance of 3049. The known best solution is shown in figure 2b with a total distance of 2579.



(a) Best Solution Found



(b) Optimal Solution

We need to note that the best solution we found largely depends on how we initialize the population. If use the greedy method as described earlier, we would have a very close solution earlier on. The bad thing is that it is very hard to jump out of local optimal. Even when we combined both the random initialization with the greedy approach. Since the greedy approach gives a much better solution than the random one, the future generation would be dominated by the genes inherited from the greedy parents. That's why we also implemented tournament selection for the second problem since we found that wheel selection has put in the system a lot of selection pressure. However, changing to tournament selection still does not help much. To break out of local optimal, we need to implement local heuristics like k-opt as proposed by [LK73]. However, we think better methods exist out there, such as linear programming that can deal with these problems more easily without setting a lot of parameters.

## REFERENCES


[Ban90] Wolfgang Banzhaf. “The “molecular” traveling salesman”. In: *Biological Cybernetics* 64.1 (1990), pp. 7–14.

[GL+85] David E Goldberg, Robert Lingle, et al. “Alleles, loci, and the traveling salesman problem”. In: *Proceedings of an international conference on genetic algorithms and their applications*. Vol. 154. Lawrence Erlbaum Hillsdale, NJ. 1985, pp. 154–159.

[LK73] Shen Lin and Brian W Kernighan. “An effective heuristic algorithm for the traveling-salesman problem”. In: *Operations research* 21.2 (1973), pp. 498–516.

[MA94] Zbigniew Michalewicz and Jarosław Arabas. “Genetic algorithms for the 0/1 knapsack problem”. In: *International Symposium on Methodologies for Intelligent Systems*. Springer. 1994, pp. 134–143.

[Pis99] David Pisinger. “Core problems in knapsack algorithms”. In: *Operations Research* 47.4 (1999), pp. 570–575.