Engineering and Technology Journal e-ISSN: 2456-3358

Volume 09 Issue 07 July-2024, Page No.- 4412-4426

DOI: 10.47191/etj/v9i07.10, I.F. – 8.227

© 2024, ETJ

0-1 Knapsack Problem Solving using Genetic Optimization Algorithm

Mubarak Altamimi[1], Nehad Ramaha[2]

[1,2]Karabuk University, Department of Computer Engineering, Karabük, Türkiye

ABSTRACT: A 0-1 knapsack problem with m constraints is known as the 0-1 multidimensional knapsack problem, and it is challenging to solve using standard techniques like branch and bound algorithms or dynamic programming. The goal of the Knapsack problem is to maximize the utility of the items in a knapsack while staying within its carrying capacity. This paper presents a genetic algorithm with Python code that can solve publicly available instances of the multidimensional knapsack problem in a very quick computational time. By identifying the significant genes, the attribute reduction method that uses the rough set theory reduces the search space and guarantees that useful information is not lost. To regulate convergence, the algorithm makes use of many additional hyperparameters that can be adjusted in the code.

KEYWORDS: Optimization Algorithms, 0-1 Knapsack Problem, Genetic Algorithm, Dynamic programming, Python.

INTRODUCTION

The goal of the combinatorial optimization problem known as the Knapsack problem is to maximize the utility of the items in a backpack while staying within its carrying capacity [1]. Swarm intelligence algorithms may be able to solve the classic NP-hard 0-1 knapsack problem (KP) [2]. The aim is to pack a backpack as efficiently as possible such that the overall weight is less than or equal to the capacity and each item's weight and profit value are compared to the capacity [3]. Stochastic approaches are increasingly being used to address the classic combinatorial optimization problem known as the Knapsack problem, first presented by Dantzig in 1950 [4]. previously, a series of mathematical approaches have been given to solve the 0-1Knapsack problem using metaheuristic optimization algorithms. they used famous optimization algorithms such as the Ant Colony Algorithm, Genetic, Greedi, Brute Force, and other optimization algorithms. In addition, some studies based their methodology on a modified hybrid metaheuristic optimization algorithm to solve the 0-1 knapsack problem, basing their work on pure mathematical methods and models [2-6]. This study aims to provide a comprehensive review of the literature and previous research on solving the 0-1 knapsack problem using optimization algorithms, in addition to contributing to how genetic algorithms can be used to solve the problem using advanced dynamic programming methods.

1.1. 0-1 Knapsack Problem

Intelligent optimization techniques such as the meta-heuristic optimization algorithm have split the traditional 0-1 knapsack problem into two and have shown success in a variety of domains, including national security, engineering technology, industrial management, and economic planning. One may reduce a significant number of engineering optimization issues to KP problems. We thus concentrate on 0-1 KP. You are requested to locate a collection of objects inside the knapsack, given a set of I $=\{i_1, i_2, i_3, \ldots\ldots, i_n\}$ and the knapsack's maximum capacity [5]. Every item has a distinct weight (Wi) and value (Pi) [5]. By using a mathematical model, the maximum capacity of the knapsack is guaranteed not to surpass the whole value of the products, hence guaranteeing that the overall weight of the items does not surpass the capacity [5]. The model in mathematics is:

$$\text{Maximize } f(x) = \sum_{i=1}^{n}(\text{pixi}), \qquad (1)$$

$$\text{sbjct to} \begin{cases} \sum_{i=1}^{n} \text{wixi} \le C \\ \text{xi} = 0 \text{ or } 1, i = 1, .. N \end{cases} \qquad (2)$$

The carrying weight of the knapsack is denoted by C in the formula, along with the number of items (n), the value (pi) and the weight (wi) of the No.i item (xi), and a choice variable (between 0 and 1).

1.2. Scenario of problem and problem statement

When robbing a store, a robber can fit the maximum weight of **W** into his backpack. There are n items, each weighing wi, and choosing this one will yield a profit of **p**i. What goods ought the burglar to take***?***

Assume that item i has the highest number in an ideal solution **S** for W dollars. Then, the optimal solution for W - wi dollars is **S'** = **S** − {i}, and the value of the solution **S** is **V**i plus the subproblem's value***.*** This fact can be expressed using the formula below: Define the answer for items 1, 2,..., i and the maximum weight w as **X**[i, w].

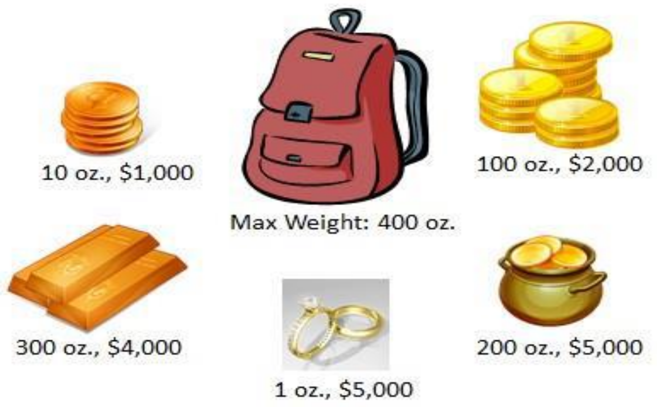


Fig. 1. Knapsack proplem idea[5].

1.3. The Study and Implement Algorithms

The 0-1 KP has been solved using a variety of methods, including both exact and approximative algorithms. It is difficult to solve this NP-hard problem with polynomial time complexity, though [6]. In contrast to exact algorithms, approximate algorithms are widely employed to solve NP-hard problems and provide reasonable solutions in a reasonable amount of time.

1.3.1. Distributed\Parallel Techniques computing for Solving 0-1Knapsack Problem

Knapsack problems are excellent candidates for parallelization because they inherently split the problem into smaller, independent sub-problems. Here's how to use distributed and parallel computing methods to solve knapsack difficulties more quickly:

1.3.1.1. Techniques for Distributed Computing:

- Master-Worker approach: With this method, multiple "worker" nodes are assigned to distinct sub-problems within the knapsack issue by a central "master" node. Worker nodes independently solve the subproblems and report their results back to the master, which assembles the entire answer.
- MapReduce: This distributed computing system can be used for variants in the knapsack problem wherein item counts and weights are independent. While the "map" function distributes tasks among worker nodes, the "reduce" function aggregates partial solutions to select the best one.

1.3.1.2. Parallel Computing Techniques

- Dynamic Programming Parallelization: It is possible to parallelize the conventional dynamic programming approach for knapsack problems by dividing the computations over multiple cores or processors. This can be achieved by splitting the dynamic programming field and performing simultaneous calculations for each sub-table.
- Branch-and-Bound Parallelization: This approach simultaneously explores multiple branches of the solution space. The best answer may be found more quickly if each processor can work on a separate branch.
- Task-Based Parallelism: The knapsack problem can be broken down into smaller jobs, such as deciding which specific objects to include. These jobs are then divided over multiple cores and executed concurrently to process data faster.

1.3.1.3. Distributed and Parallel Computing Advantages

- Scalability: As the size of the issue increases, more processing units can be added to preserve the system's efficiency.
- Speedup: By distributing calculations across multiple processing units, these solutions can significantly reduce the time required to solve complex knapsack problems.

1.3.2. Approximation and heuristics Algorithms for solving 0-1Knapsack problem.

As members of the NP-hard problem class, knapsack issues are notoriously difficult to solve exactly for large cases. This suggests that it can take longer to find the ideal answer as the problem's complexity rises. However, several approaches that employ approximation algorithms and heuristics can yield precise results much faster.

- Greedy Algorithm: these are straightforward and effective techniques that fill the knapsack one item at a time until it is full. Sorting products according to their highest profit-to-weight ratio is a popular tactic. Despite not always being the best, greedy algorithms frequently locate workable solutions quickly.
- Relaxation using Linear Programming (LP): This method rewrites the knapsack problem as a linear program where items may be partially included. There's not much utility for fractional knapsacks, but the solution provides a lower bound on the optimal value, even though it might not be immediately helpful. This lower bound can help evaluate various heuristic solutions.
- Dynamic programming: a process wherein answers to smaller subproblems are assembled to address a larger problem. For some backpack versions, it might

function well, but the memory requirements might go up as the issue size rises.

- Metaheuristics: these are more advanced approaches that use iterative search strategies to find the solution space. Among these are simulated annealing, genetic algorithms, and ant colony optimization. These methods provide good solutions for large problems, while careful parameter tweaking may be required.
- Large Neighbourhood Search (LNS): This metaheuristic method explores a wider range of possibilities by upending established solutions. Removing and adding items iteratively can yield significant benefits.
- Guaranteed-Performance Approximation Algorithms: these algorithms compute guaranteed answers within a given factor (e.g., within 1-ε of the best result) in polynomial time. For large-scale problems, however, their approximation factor might not be ideal, and benefit from guaranteed performance and fast runtime.
- Brute Force Algorithm: are precisely what they sound like: simple approaches to problem-solving that rely on a computer's raw capability and exploring every option rather than sophisticated strategies to increase efficiency.

1.3.3. Genetic Algorithm for 0-1knapsack problem

For big cases, NP-hard issues like knapsack problems might be difficult to solve, although heuristics and approximation techniques can yield faster, more precise solutions. Utilizing computer simulations, genetic algorithms identify potential solutions from a population of individuals called chromosomes to optimize problems. Natural selection and mutation produce new populations by assessing adaptation, choosing individuals, and starting with a random population. Evolution happens from generation to generation[7]. The fundamental genetic algorithms begin with an initial population and proceed to create additional populations by natural selection, crossover, and mutation processes. These populations are then updated continuously until the best possible solutions are identified. The stages involved in the calculation for the knapsack problem are as follows:

1. Coding: I have created a chromosomal encoding approach for the knapsack issue based on its paradigm. An n-bit binary string x contains the encoded data for n objects. When x[i] = 1, object i has been put in the knapsack; otherwise, it implies that object i is not in the knapsack. As an illustration, the number 1010 denotes a solution, which indicates that only items 1 and 3 are packed in a backpack and not the remaining items[8].
2. Creating the initial population: in this scenario, the population size is set to n, meaning that population A is made up of n individuals, and A[i] can be created at random.
3. Calculating population fitness: The following formula is used to determine each member's fitness within a population:

$$\Big\{ T = \sum w[i]x[i] \tag{3}$$

$$\text{FTNSS} = \begin{cases} \sum V[i]X[i, \text{if } T \leq W \\ \sum V[i]x[i] - \alpha * (T - W), \text{if } T \geq W \end{cases}$$
(4)

   Equation (3) represents the fitness penalty function, with α > 1.0. For this instance, we'll choose α = 2 [8].
4. Choice: To calculate the population of the following generation, I use a likelihood proportional to fitness. The steps of the process are as follows:
   a - Determine the total fitness of every member in the population, denoted as Σ f (A[i]), where i ranges from 1 to n.
   b - Second, for each i = 1, 2,..., n, the relative fitness for each person is computed as p(A[i] = f (A[i])/Σ f (A[i]).
   c - For any probabilistic value, the entire sum of the probabilistic values equals 1.
   d - Following the generation of a random number from 0 to 1, the amount of unknown numbers that take place in the aforementioned probabilistic zones determines how many times an individual is chosen.
5. Intersections: A cross-cutting technique with a single point is used. First, groups are matched at random. Next, a random location is chosen for the crossing point. After that, a few genes that connect the linked chromosomes are exchanged.
6. Mutation: To carry out the mutation operation, I employ the basic mutation technique. First, each person's location regarding gene variation was ascertained. Then, based on a specific likelihood, the mutation point's initial genetic value is inverted [8].

PSEUDO CODE FOR GENETIC ALGORITHM

```
t := 0; // start with an initial time
initpopulation P (t); // initiate a population of individuals typically at random.
evaluate P (t); // assess the fitness of each initial individual in the population.
while not done do // checks for the termination criteria.
t := t + 1; // extend the timer.
P' := selectparents P (t); // select a subpopulation to produce offspring from.
recombine P' (t); // reassemble a chosen parent's "genes".
mutate P' (t); // random perturbation of the mated population.
evaluate P' (t); // evaluate its increased fitness.
P := survive P,P' (t); // select the survivors based on their level of fitness.
```

od
end GA. [9].

Related Work

As more absolute quantity equals more time required, the knapsack problem contributes to the NP problem, which raises the issue of temporal complexity. Using genetic algorithms could provide advantages over the standard themes described above and result in more optimal and efficient solutions. In[1], authors perform 0-1 knapsack tasks based on genetic algorithms and use the results to determine the fitness and incentive for the next generation of people who have the innate desire to find the best possible arrangement. In[2] An author-modified version of the HRO algorithm is suggested for the challenging large-scale 0-1 KP. In the renewal stage, a dynamic step is included to balance the stages of exploration and exploitation. In[3] A novel cognitive behavior optimization algorithm (COA) was applied by the author to solve large-scale 0-1 knapsack problems. Four large-scale knapsack problems were solved using COA. In[4], the authors present a novel reduction technique for the 0-1 knapsack problem (0- 1 KP) and a better mutations operator (IMO) based on the premise NP 6= P. In[5], For KP01, the author suggests a novel population-based SA (PSA) and evaluates it against the current approaches. In[6], the author put forth a distribution estimation-based hybrid harmony search technique. To enhance algorithm searching performance, a fixed improvisation procedure is introduced. In[7], the researcher creates a model update algorithm using particle swarm optimization (PSO) and the eigenvector difference approach. The outcomes demonstrate the increased accuracy of the suggested ED-PSO model updating approach, which is anticipated to be more useful for bridge finite element model updating study. In[8], the GA model's solution revealed that no combination could provide the precise weight or capacity that the 35-kilogram bag could hold, but 34 kg and 36 kg are the range that the solution model could suggest. In[9], researchers show that in multidimensional knapsack problems, the FAGA model performed rather well. In[10], the author used GSA & GA Algorithms and made sure multidimensional knapsack problems were solved using the hybrid GSA-GA approach. In[11], the authors contrast the Greedy technique and the Dynamic Programming approach as two methods for solving the KP. Greedy is superior to DP in terms of runtime and space requirements, but DP performs better in terms of the optimized solution. In[12], modeling to compare instances that were created at random and for which the metaheuristics were implemented in the cloud, Google Colaboratory, and Python programming. In[13] suggests using the recently created Gaining Sharing knowledge-based optimization algorithm (GSK) in a unique binary form to handle binary optimization challenges. In[14] To compare the GBLSO algorithm with the Binary Bat Algorithm (BBA) and Discrete Binary Particle Swarm Optimisation (DPSO) algorithm, ten common MKP cases were simulated. In[15] A series of numbers known as "chromosomes" make up the population, and the sequence of numbers that makes up a chromosome is known as a "gene.". In[16], the authors suggest using a Binary Genetic Algorithm (BGA) and a third-party archive. Additionally, a kind of binary local search is used in the suggested BGA algorithm. In[17] the following five meta-heuristic algorithms have recently been proposed: gradient-based optimizer (GBO), golden eagle optimizer (GEO), red fox search optimizer (RFSO), horse herd optimization algorithm (HOA), and bonobo optimizer (BO). In[18], a heuristic algorithm An optimization technique based on group theory is suggested to solve KPS using RA-GTOA. In[19], combinatorial optimization issues, such as the traveling salesman problem, set-covering problem, least spanning tree problem, knapsack problem, and bin-packing problem, as well as genetic algorithms. In[20] the Bayesian Multiploid Genetic Algorithm is responsible for solving the well-known Multidimensional Knapsack Problem (MKP). An enormous edge in resolving the given problem comes from making use of the relationships between the variables.

Table 1. Literature Review For Most Important Papers

No.	**Paper title**	**publishing**	**Method that used**	**Results**
[1]	Bansal, A., et al.,2021	Esaychair	Genetic algorithm GA	In GA, all chromosomes are rearranged to fit under the highest possible weight.
[2]	Shu, Z., Ye, et al.,2024	arXiv	HRO algorithm, Python 3.6	By harmonizing the periods of exploration and exploitation, the dynamic step approach raises the quality of solutions.
[3]	Yang, Y., et al.,2024	arXiv	UBMP & GA	Large-scale NP-hard issues are the main use case for Gas, the IMO typically performs better than the MO.
[4]	Wei, Y., et al.,2020	IEEE	COA Algorithm	The COA demonstrates the effectiveness and viability of COA in the extensive 0-1 KP.
[5]	Moradi, N., et al.,	Springer	SSAs and PSA algorithms	Of all the SA-based solutions, PSA is the most effective optimization strategy for KP01.

[6]	Liu, K., et al.,2022	Hindawi	HHSEDA Algorithm	HHSEDA is stable and has good optimization potential when it comes to addressing the 0-1 knapsack problem.
[7]	Wang, C., et al.,2023	Extrica	FEMU Algorithm	ED-PSO model updating approach is more accurate, bridge finite element simulation updating research is anticipated to benefit from its use.
[8]	Okwu, M., et al,2020	biblioteka nauki.pl	Genetic algorithm GA	Finding roughly optimal solutions to an NP problem is made achievable by the GA technique, decreasing the KP's complexity to linear.
[9]	Nand, R., et al,2019	IEEE	FA & GA Algorithms	In multidimensional knapsack problems, the FAGA model performed rather well.
[10]	Kumar Gupta, 2018	IEEE	GSA & GA Algorithms	Multidimensional knapsack problems are solved using the hybrid GSA-GA approach.
[11]	AlEtawi, et al,2020	IEEE	GA & DP algorithms	Greedy is superior to DP in terms of runtime and space requirements, but DP performs better in terms of the optimized solution.
[12]	do Vale, et al.,2023	Springer	ANOVA MODEL	The genetic algorithm produced more satisfactory outcomes than the other metaheuristics.
[13]	Agrawal, et al.,2021	Springer	NBGSK model Algorithm	NBGSK and PR-NBGSK provide superior efficacy and efficiency for convergence, robustness, and precision.
[14]	Yang, Y., et al.,2021	joca.cn	DPSO\BBA & GBLSO Algorithm	The GBLSO algorithm is powerful for addressing MKP problems. It has strong robustness, high optimization accuracy, and good convergence efficiency.
[15]	Saraswat, et al.,2021	Springer	GA Algorithm	A population's fitness value for each chromosome affects the likelihood that that specific chromosome will survive in the following generation.
[16]	Kabadurmus, et al.,2021	Springer	BGA Algorithm	BGA is a more efficient solution for the BOMDKP.
[17]	Abdel-Basset, et al.,2022	Elsevier	GBO, RFSO, HQA, BO Algorithms	BIRFSO's competitiveness for the remaining cases and its superiority for those with dimensions larger than 500.
[18]	He, Y., et al.,2024	Elsevier	RA-GTOA Algorithm	An effective algorithm for resolving KPS is RA-GTOA.
[19]	Gen, M., et al.,2023	Springer	GA Algorithm	GA is an efficient algorithm for solving KPS.
[20]	Gazioğlu, et al.,2022	Dergipark	BMGA Algorithm	An enormous edge in resolving the given problem comes from taking use of the relationships between the variables.

The Proposed Method

By growing a population of potential solutions to a problem, each having altered attributes and typically expressed in binary, a Genetic algorithm (GA) is an algorithm that mimics natural selection. Each generation of evolution is an iterative process that begins with a random population. A new generation is created by selecting fit individuals and modifying their genomes. When the desired level of fitness is attained or a maximum number of generations is achieved, the algorithm stops

.

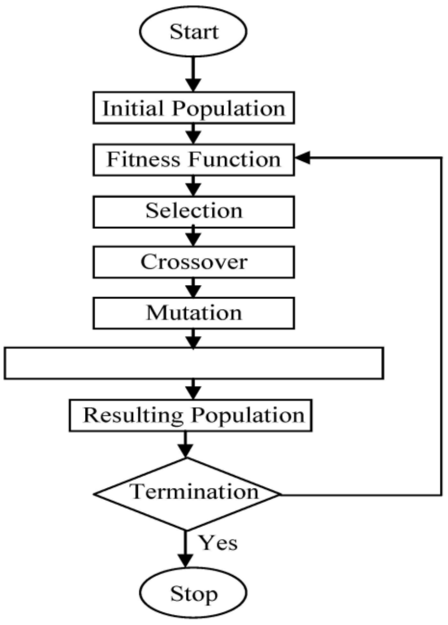


Fig. 1. A methodology of genetic algorithm for knapsack problem [11].

The majority of genetically altered procedures (GAs) rely on multiple factors, including chromosomal populations, fitness-based selection, progeny crossover, and random mutation of offspring [12].

Chromosomes: The space of potential solutions is represented by the chromosomes in GAs. Different kinds of chromosomal encodings exist. We employ binary encoding for the Knapsack problem, in which each chromosome is represented by a string of bits, either 0 or 1.

(5)

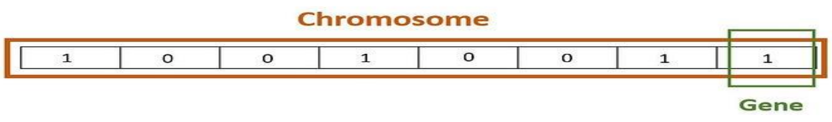


Fig. 2. Chromosome initialization gene [11].

Initialize chromosom like this operation code:

```
#generating chromosome with probability of 1's
import random
def generate_chromosome(N, w, L, p): #N choromosome_size, weight, #limit, probability
  score = 0
  g = np.zeros(N) #verify if vector is 64 or 65
  for i in range(len(g)):
    prob = random.uniform(0, 1)
    if prob < p:
      g[i] = 1
    else:
      g[i]
```

$$g_i = \begin{cases} 1 & \text{iff} \quad \psi < p \\ 0 & \text{otherwise} \end{cases}$$

for all $i \in < 0, N-1 >$ where $N = |\mathbf{g}|$.

```
= 0
  for c in range(N):
    score = score + np.sum(w[c]*g[c])
```

```
 if score <= L:
   pass
 return g
```

Encoding og chromosoms: One way to represent a chromosome is with an array whose size is the same as the number of entries (in our case, three). This array's elements each indicate whether an item is in the knapsack ('1') or not ('0') [I]. Take this chromosome, for instance:

$$r_i = \frac{f_j}{F} + r_{i-1}; r_{-1} = 0, F = \sum_{k=0}^{N-1} f_k$$

Shows that the knapsack contains the first and third items (A and C).
Fitness: Chromosomes' code and problem-solving efficiency, such as the chromosome's total profit in the 0-1 knapsack problem, are calculated by GAs using fitness functions to assign scores to each chromosome.
(6)

A	B	C
1	1	0

To impliment function of fitness letus implement this task code:

```
#fitness function for each chromosome
def fitness(w, c, L, g): #weight, cost, weight_limit, chromosome score = 0
score1 = 0
for i in range(len(w)):
score = score + np.sum(w[i]*g[i])
if score > L: f = 0
else: for i in range(len(w)):
score1 = score1 + np.sum(c[i]*g[i])
f = score1
return score1
```

$$\text{fitness}(\mathbf{g}) = \begin{cases} 0 & \text{iff} \quad \sum_{i=0}^{N-1} w_i g_i > L \\ \sum_{i=0}^{N-1} c_i g_i & \text{otherwise} \end{cases}$$

A chromosome's fitness, which comprises only A and B, is calculated by combining their profits, yielding a fitness of 7.
Selection: Fitness plays a key role in the selection process, as more fit chromosomes are more likely to replicate. Like natural organism survival, the number of chosen chromosomes is proportionate to the population size, ensuring a stable size for every generation. This equation shows the se
(7)
This code shows how we can implement the selection step:

```
#roulette selection based on fitness score
Choose the population and fitness, use probability and random selection in roulette, and choose the chromosome according to the parent-size specification.
 fitness = fitness_score
 total_fit = sum(fitness)
 relative_fitness = [f/total_fit for f in fitness]
 cum_probs = np.cumesum(relative_fitness)
 roul = np.zeros((parents, len(pop[0]))) # matrix's form depending on parent size.

 for i in range(parents):
   r = random.uniform(0, 1)
   for ind in range(len(pop[:, 0])): # number of population entries
```

```
  if cum_probs[ind] > r:
    #print(r) '''for debugging'''
    #print(ind)
    roul[i] = pop[ind]
    break
return roul
```

Crossover: Through a biological process called crossover, chromosomal fragments from both parents are combined to generate offspring[13]. A locus is chosen at random, and two chromosomes' subsequences are switched.
(8)

$$r_k = \begin{cases} b_k & k <= i \wedge \psi < p \\ a_k & \text{otherwise} \end{cases}$$

Parent1	100	0111
Parent2	111	1000
Ofspring1	100	1000
Ofspring2	111	0111

This code implement a crossover step and operation:

```
def crossover(a, b, p): #a=chromosome 1, b=chromosome 2,
#p = probability for crossover
  ind = np.random.randint(0, 64)
  r = random.uniform(0, 1)
  if r < p:
    c1 = list(b[:ind]) + list(a[ind:]) #since array were having shape issues, converting to lists
    c1 = np.array(c1)
    c2 = list(a[:ind]) + list(b[ind:])
    c2 = np.array(c1)
  else:
    c1 = a
    c2 = breturn c1, c2 #returning the crossover childs
```

Mutation: As demonstrated by chromosomes having a mutation point at location 2, genetic algorithm mutation preserves genetic variety by flipping bits from 1 to 0 or 0 to 1.
(9)

Orginal chromosom	1**0**00111
Mutatute chromsom	1**1**00111

$$r_k = \begin{cases} b_k & k <= i \wedge \psi < p \\ a_k & \text{otherwise} \end{cases}$$

A **0** at two position will be flip to 1after one time matuate.
This code shows how we can implement the mutation operation in the genetic algorithm for 0-1 knapsack problem solving:

```
#mutattion of bits from 1 to 0 and 0 to 1 based on probability
def mutation(g, p):
  N = len(g)
  m = np.zeros(len(g)) #mutated chromosome
  for i in range(N):
    d = g[i]
    r = random.uniform(0, 1)
    if g[i] == 1.0 and r < p:
      m[i] = 0
    elif g[i] == 0.0 and r < p:
```

```
    m[i] = 1
  else:
    m[i] = d
return m
```

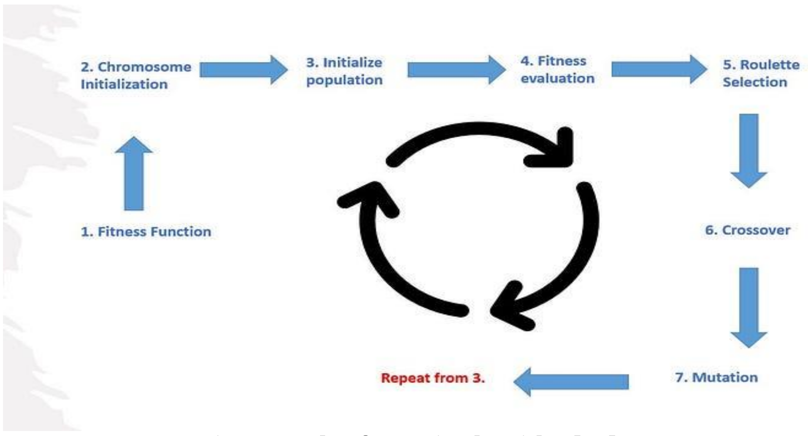


Fig. 3. Cycle of genetic algorithm[13].

The rationale behind genetic algorithms is similar to the idea that life is an evolution of generations, with the only distinction being that excellent generations are retained. In contrast "bad generations" that may not have adequate fitness functions are discarded [21]. First, the program generates 100 random arrays of size 5. Binary values 0 and 1 make up this array. Following initialization, the selection function chooses the top 50 arrays based on their fitness. The approach that verifies each array for weight and profit correspondence is the most fit [21]. For every index of the parent array, which is 1, those two arrays hold the weight and the profit. For optimization, the arrays are now randomly generated via the mutation mechanism. In this instance, my approach uses a range of 0,1 to produce random values for each array's index. The index is then switched from 0 to 1 or 1 to 0, depending on whether the value is smaller than 0.5 [22]. The next technique used in the running time is the crossover, which creates children by combining two parents (arrays)[22]. This method aims to produce 100 new parents for the algorithm's subsequent iterations and identify the ideal profit margin. When the ideal value is discovered, the program ends if not, it keeps creating new offspring until it does, which in this example is [0, 1, 1, 1, 0].

Explanation of Genetic Algorithm Procedures

1. [Start] Generate a random population of n chromosomes (suitable solutions for the problem)
2. [Fitness] Evaluate the fitness f(x) of each chromosome x in the population
3. [New population] Create a new population by repeating the following steps until the new population is complete:
   I. [Selection] Select two parent chromosomes from a population according to their fitness (the better fitness, the bigger the chance to be selected)
   II. [Crossover] With a crossover probability cross over the parents to form a new offspring (children). If no crossover was performed, the offspring is an exact copy of the parents.
   III. [Mutation] With a mutation probability mutate new offspring at each locus (position in chromosome).
   IV. [Accepting] Place new offspring in a new population
4. [Replace] Use the newly generated population for a further run of the algorithm
5. [Test] If the end condition is satisfied, stop, and return the best solution in the current population

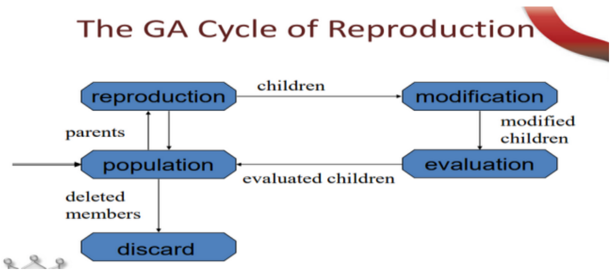


Fig. 4. The Cycle reproduction of genetic algorithm [24].

1. Encoding

Encoding of chromosomes is one of the problems, when you are starting to solve problem with GA. Encoding very depends on the problem.

1.1 Binary Encoding

• Binary encoding is the most common, mainly because first works about GA used this type of encoding.

• In binary encoding every chromosome is a string of bits, 0 or 1

1.2 Permutation Encoding

• Permutation encoding can be used in ordering problems, such as travelling salesman problem or task ordering problem.

• In permutation encoding, every chromosome is a string of numbers, which represents number in a sequence.

1.3 Value Encoding

- Direct value encoding can be used in problems, where some complicated value, such as real numbers, are used. Use of binary encoding for this type of problems would be very difficult.
- In value encoding, every chromosome is a string of some values. Values can be anything connected to problem, form numbers, real numbers or chars to some complicated objects.

2. Selection

The problem is how to select these chromosomes. According to Darwin's evolution theory the best ones should survive and create new offspring. There are many methods how to select the best chromosomes, for example, roulette wheel selection, Boltzman selection, tournament selection, rank selection, steady state selection

2.1 Roulette Wheel Selection

Parents are selected according to their fitness. The better the chromosomes are, the more chances to be selected they have. Imagine a roulette wheel where are placed all chromosomes in the population, every has its place according to its fitness function

2.2 Rank Selection

- The previous selection will have problems when the fitnesses differs very much. For example, if the best chromosome fitness is 90% of all the roulette wheel then the other chromosomes will have very few chances to be selected.
- Rank selection first ranks the population, and then every chromosome receives fitness from this ranking. The worst will have fitness 1, the second worst 2, and the best will have fitness N (number of chromosomes in population).

2.3 Tournament selection

Tournament selection involves running several "tournaments" among a few individuals chosen at random from the population. The winner of each tournament (the one with the best fitness) is selected for crossover. Selection pressure is easily adjusted by changing the tournament size. Weak individuals have a smaller chance of being chosen if the tournament size is larger.

Population

Chromosomes could be:

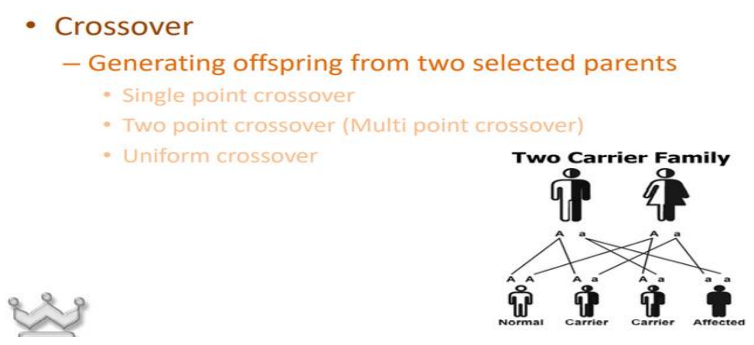
– Bit strings (0101 ... 1100)

– Real numbers (43.2 -33.1 ... 0.0 89.2)

– Permutations of element (E11 E3 E7 ... E1 E15)

– Lists of rules (R1 R2 R3 ... R22 R23)

– Program elements (genetic programming)



- Crossover
  – Generating offspring from two selected parents
    - Single point crossover
    - Two point crossover (Multi point crossover)
    - Uniform crossover

Fig. 4. Crossover Offspring Operation [24].

- Crossover rate

Crossover rate generally should be high, about 80%-95%. (However some results show that for some problems crossover rate about 60% is the best.)

- Mutation rate

On the other side, mutation rate should be very low. Best rates reported are about 0.5%-1%.

- Population size

It may be surprising, that very big population size usually does not improve performance of GA (in meaning of speed of finding solution). Good population size is about 20-30, however sometimes sizes 50-100 are reported as best.

- Some research also shows, that best population size depends on encoding, on size of encoded string. It means, if you have a chromosome with 32 bits, the population should be say 32, but surely two times more than the best population size for a chromosome with 16 bits.

- Encoding

Encoding depends on the problem and also on the size of the instance of the problem.

- Crcrossover and mutation type

Operators depend on encoding and the problem. about operators for some suggestions.

01 Knapsack Problem Dataset

A dataset has sample data for the 01 Knapsack issue. The 01 Knapsack problem involves a rucksack with a capacity of C and N items with weights and profits. The goal is to increase the overall profit from each item. A subset S of items that maximize overall profit and have a weight sum below or equal to C is the solution[23].

ınput

Knapsack Problem	
Instance	**Optimal**
ks_8a	3,924,400
ks_8b	3,813,669
ks_8c	3,347,452
ks_8d	4,187,707
ks_8e	4,955,555
ks_12a	5,688,887
ks_12b	6,498,597
ks_12c	5,170,626
ks_12d	6,992,404
ks_12e	5,337,472
ks_16a	7,850,983
ks_16b	9,352,998
ks_16c	9,151,147
ks_16d	9,348,889
ks_16e	7,769,117
ks_20a	10,727,049
ks_20b	9,818,261
ks_20c	10,714,023
ks_20d	8,929,156
ks_20e	9,357,969
ks_24a	13,549,094
ks_24b	12,233,713
ks_24c	12,448,780
ks_24d	11,815,315
ks_24e	13,940,099

Best cost table:

Permutation	**Total Cost**
(0, 1, 2, 3)	82
(0, 1, 3, 2)	83
(0, 2, 1, 3)	157
(0, 2, 3, 1)	64
(0, 3, 1, 2)	29
(0, 3, 2, 1)	128
(1, 0, 2, 3)	79
(1, 0, 3, 2)	80
(1, 2, 0, 3)	32
(1, 2, 3, 0)	64
(1, 3, 0, 2)	128
(1, 3, 2, 0)	128
(2, 0, 1, 3)	150

Iteration table:

Iteration	**Random Number Set 1**	**Random Number Set 2**
1	0.45231413352303207	0.45231413352303207
2	0.45231413352303207	0.45231413352303207
3	0.45231413352303207	0.45231413352303207
4	0.45231413352303207	0.45231413352303207

5	0.45231413352303207	0.45231413352303207
6	0.45231413352303207	0.45231413352303207
7	0.45231413352303207	0.45231413352303207
8	0.45231413352303207	0.45231413352303207
9	0.45231413352303207	0.45231413352303207
10	0.45231413352303207	0.45231413352303207
11	0.45231413352303207	0.45231413352303207
12	0.45231413352303207	0.45231413352303207
13	0.45231413352303207	0.45231413352303207
14	0.45231413352303207	0.45231413352303207
15	0.45231413352303207	0.45231413352303207
16	0.45231413352303207	0.45231413352303207
17	0.45231413352303207	0.45231413352303207
18	0.45231413352303207	0.45231413352303207
19	0.45231413352303207	0.45231413352303207
20	0.45231413352303207	0.45231413352303207
21	0.45231413352303207	0.45231413352303207
22	0.45231413352303207	0.45231413352303207
23	0.45231413352303207	0.45231413352303207
24	0.45231413352303207	0.45231413352303207

Experimental Results

This Python genetic algorithm example is quite basic. An array of random strings is evolved in the direction of the target string by this code:

```
#Represent a chromosome as a binary vector of length
500000 (5000 families, 100 days).
chromosome = [0 for i in range(500000)]
for i in range(5000):
    chromosome[i*100+best[i]-1] = 1

population = []
population.append(chromosome)

family_size_dict = data[['n_people']].to_dict()['n_people']

cols = [f'choice_{i}' for i in range(10)]
choice_dict = data[cols].T.to_dict()

N_DAYS = 100
MAX_OCCUPANCY = 300
MIN_OCCUPANCY = 125

# from 100 to 1
days = list(range(N_DAYS,0,-1))

family_size_ls = list(family_size_dict.values())
choice_dict_num = [{vv:i for i, vv in enumerate(di.values())}
for di in choice_dict.values()]

# Computer penalities in a list
penalties_dict = {
    n: [
        0,
        50,
        50 + 9 * n,
        100 + 9 * n,
        200 + 9 * n,
        200 + 18 * n,
        300 + 18 * n,
        300 + 36 * n,
        400 + 36 * n,
        500 + 36 * n + 199 * n,
        500 + 36 * n + 398 * n
    ]
    for n in range(max(family_size_dict.values())+1)
}

def cost_function(prediction):
    penalty = 0

    # We'll use this to count the number of people scheduled
each day
    daily_occupancy = {k:0 for k in days}

    # Looping over each family; d is the day, n is size of that
family,
    # and choice is their top choices
    for n, d, choice in zip(family_size_ls, prediction,
choice_dict_num):
        # add the family member count to the daily occupancy
        daily_occupancy[d] += n

        # Calculate the penalty for not getting top preference
        if d not in choice:
            penalty += penalties_dict[n][-1]
        else:
            penalty += penalties_dict[n][choice[d]]

    # for each date, check total occupancy
    #  (using soft constraints instead of hard constraints)
    for v in daily_occupancy.values():
        if (v > MAX_OCCUPANCY) or (v < MIN_OCCUPANCY):
            penalty += 100000000

    # Calculate the accounting cost
    # The first day (day 100) is treated special
    accounting_cost = (daily_occupancy[days[0]]-125.0) / 400.0 * daily_occupancy[days[0]]**(0.5)
    # using the max function because the soft constraints might allow occupancy to dip below 125
    accounting_cost = max(0, accounting_cost)

    # Loop over the rest of the days, keeping track of previous count
    yesterday_count = daily_occupancy[days[0]]
    for day in days[1:]:
        today_count = daily_occupancy[day]
        diff = abs(today_count - yesterday_count)
        accounting_cost += max(0, (daily_occupancy[day]-125.0) / 400.0 * daily_occupancy[day]**(0.5 + diff / 50.0))
        yesterday_count = today_count

    penalty += accounting_cost

    return penalty
best = -1
best_val = 105163.8446075958
for i in range(20):
    print(i)
    population = selection(population, 25, 5)
    population = reproduction(matrix, population, 50, 0.25, 10)
    ind, val = epoch_optimal(population)
    print('Min on epoch: ', str(val))
    if best_val > val:
        best_val = val
        best = ind
```

Explanation for the code:
Import libraries: numpy is the most commonly used numerical library function.
Define function: Cost function which is used to make several decision variables.
Define variable: These represent the number of data to be processed per batch or the number of decision variables.
Maxit: This involves the number of iterations of the population sample or the number of iterations.
Sigma: This evaluates a certain expression many times, with slightly different variables, and returns the sum of all those expressions or step size of mutation.
Plt.plot: It is the library used to generate or plot the x and y axis graphs.
Var num_children: It is a variable that makes sure it always has an even number.
Beta: the measure of risk/volatility of a stock or iteration.

The fittest individuals are chosen to reproduce through mutation from an arbitrary number of strings created by this code. The fitness function determines how many characters in the sentence fit the target string. Once a match is discovered, the evolution goes on. Complexity meticulous strategy and parameter adjustment are hallmarks of real-world genetic algorithms.

Results return per 100 iteration

```
Iteration 0: Best Cost = 11.953159210518818
Iteration 100: Best Cost = 0.2557520472002101
Iteration 200: Best Cost = 0.2557520472002101
Iteration 300: Best Cost = 0.11281281227221135
Iteration 400: Best Cost = 0.11281281227221135
```

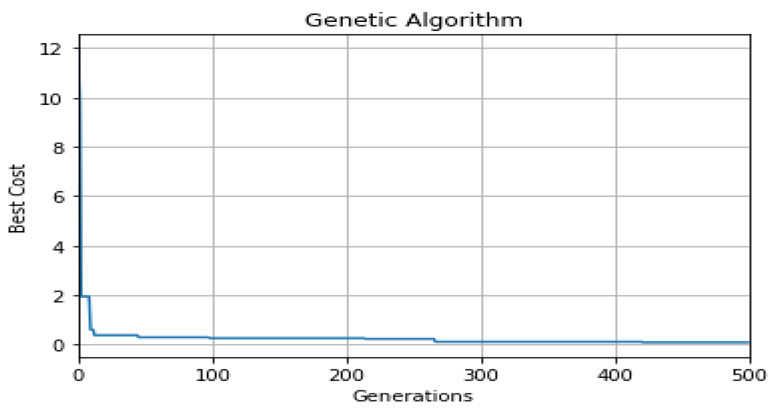


Fig. 4. Histogram of genetic algorithm results with knapsack problem solving [Jupyter Python].

The genetic algorithm is a powerful optimization technique widely applied to solve various combinatorial optimization problems, including the knapsack problem. In the context of the knapsack problem, the genetic algorithm offers a unique and effective approach to finding near-optimal solutions.

Challenges

The genetic algorithm offers a powerful approach to solving the knapsack problem, challenges such as representation and encoding, constraint handling, fitness evaluation, premature convergence, scalability, and multiple objectives need to be carefully addressed. Overcoming these challenges requires thoughtful algorithm design, parameter tuning, and the incorporation of efficient techniques from the field of evolutionary computation.

CONCLUSION

Numerous fields, such as resource allocation, material cutting and packaging, and energy management, use knapsack problems. While approximation algorithms prioritize speed, exact algorithms provide optimal answers. Research guarantees that instruments and methods for resolving increasingly complicated variants of knapsack problems will continue to progress. These improvements result in increased productivity, faster decision-making, and lower resource allocation and logistics costs. They are helpful in industries like supply chain management and big data because they enable handling larger, more complicated problems. More adaptable solutions can be obtained by fusing machine learning approaches with conventional knapsack problem-solving strategies. An effective search engine ought to be universal, with a heuristic included to provide the algorithm with helpful guidance. Running the approach on larger instances for which optimal solutions exist could be the focus of future development.

REFERENCES

1. Bansal, A., Gadia, H., Dhanusha, S., & Pandey, A. (2021). Solving 0-1 Knapsack Problem using Genetic Algorithm.
2. Shu, Z., Ye, Z., Zong, X., Liu, S., Zhang, D., Wang, C., & Wang, M. (2022). A modified hybrid rice optimization algorithm for solving 0-1 knapsack problem. Applied Intelligence, 52(5), 5751-5769.
3. Yang, Y. (2024). An upper bound of the mutation probability in the genetic algorithm for general 0-1 knapsack problem. arXiv preprint arXiv:2403.11307.
4. Wei, Y., & Luo, Q. (2020, March). Cognitive Behavior Optimization Algorithm Application for Large-scale Knapsack Problem. In 2020 IEEE International Conference on Artificial Intelligence and Information Systems (ICAIIS) (pp. 179-183). IEEE.
5. Moradi, N., Kayvanfar, V., & Rafiee, M. (2022). An efficient population-based simulated annealing algorithm for 0–1 knapsack problem. Engineering with Computers, 38(3), 2771-2790.
6. Liu, K., Ouyang, H., Li, S., & Gao, L. (2022). A hybrid harmony search algorithm with distribution estimation for solving the 0-1 knapsack problem. Mathematical Problems in Engineering, 2022.
7. Wang, C., Li, D., Kaewniam, P., Wang, J., & Al Hababi, T. (2023). An ED-PSO model updating algorithm for structure health monitoring of beam-like structures. Journal of Measurements in Engineering, 11(3), 358-372.
8. Okwu, M., Otanocha, O. B., Omoregbee, H. O., & Edward, B. A. (2020). Appraisal of genetic algorithm and its application in 0-1 knapsack problem. Journal of Mechanical and Energy Engineering, 4(1), 39-46.
9. Nand, R., & Sharma, P. (2019, December). Iteration split with Firefly Algorithm and Genetic Algorithm to solve multidimensional knapsack problems. In 2019 IEEE Asia-Pacific Conference on Computer Science and Data Engineering (CSDE) (pp. 1-7). IEEE.
10. Gupta, I. K. (2018, March). A hybrid GA-GSA algorithm to solve multidimensional knapsack problem. In 2018 4th International Conference on Recent Advances in Information Technology (RAIT) (pp. 1-6). IEEE.
11. Al Etawi, N. A., & Aburomman, F. T. (2020). 0/1 KNAPSACK PROBLEM: GREEDY VS. DYNAMIC-PROGRAMMING. Int J Adv Eng Manag Res, 5(2), 1-10.
12. do Vale Pereda, M., Scarpin, C. T., Junior, J. E. P., Puhl, C., & Ferrer, L. W. U. (2023). Comparison of Metaheuristics in Solving the Knapsack Problem: An Experimental Analysis. Revista de Gestão Social e Ambiental, 17(9), e03814-e03814.
13. Agrawal, P., Ganesh, T., & Mohamed, A. W. (2021). Solving knapsack problems using a binary gaining sharing knowledge-based optimization algorithm. Complex & Intelligent Systems, 1-21.
14. Yang, Y., Liu, S., & ZHOU, Y. (2020). Greedy binary lion swarm optimization algorithm for solving multidimensional knapsack problem. Journal of Computer Applications, 40(5), 1291.
15. Saraswat, M., & Tripathi, R. C. (2021). Solving Knapsack Problem with Genetic Algorithm Approach. In Mathematical Modeling and Computation of Real-Time Problems (pp. 169-177). CRC Press.

16. Kabadurmus, O., Tasgetiren, M. F., Oztop, H., & Erdogan, M. S. (2021). Solving 0-1 bi-objective multi-dimensional knapsack problems using binary genetic algorithm. Heuristics for Optimization and Learning, 51-67.
17. Abdel-Basset, M., Mohamed, R., Elkomy, O. M., & Abouhawwash, M. (2022). Recent metaheuristic algorithms with genetic operators for high-dimensional knapsack instances: A comparative study. Computers & Industrial Engineering, 166, 107974.
18. He, Y., Wang, J., Liu, X., Wang, X., & Ouyang, H. (2024). Modeling and solving of knapsack problem with setup based on evolutionary algorithm. Mathematics and Computers in Simulation, 219, 378-403.
19. Gen, M., & Lin, L. (2023). Genetic algorithms and their applications. In Springer handbook of engineering statistics (pp. 635-674). London: Springer London.
20. Gazioğlu, E. (2022). Solving Multidimensional Knapsack Problem with Bayesian Multiploid Genetic Algorithm. Journal of Soft Computing and Artificial Intelligence, 3(2), 58-64.
21. Wang, R., & Zhang, Z. (2021). Set theory-based operator design in evolutionary algorithms for solving knapsack problems. IEEE Transactions on Evolutionary Computation, 25(6), 1133-1147.
22. Zhang, X., Qi, F., Hua, Z., & Yang, S. (2020, April). Solving billion-scale knapsack problems. In Proceedings of The Web Conference 2020 (pp. 3105-3111).
23. P. T. Pantzan, GitHub - Pantzan/KnapsackGA: Knapsack Problem solved using Genetic optimization algorithm, (2020).
24. Baş, E. (2023). Binary aquila optimizer for 0–1 knapsack problems. Engineering Applications of Artificial Intelligence, 118, 105592.
25. William, I. O., & Altamimi, E. M. (2024). Hierarchical Long Short-Term Memory (LSTM) Model for News Sentiment Analysis.