

A PROJECT REPORT

on

“Implementation of 0-1 knapsack problem using PSO algorithm”

Submitted to

KIIT Deemed to be University

In Partial Fulfillment of the Requirement for the Award of

**BACHELOR’S DEGREE IN
COMPUTER SCIENCE & ENGINEERING**

BY

Sakshi Upadhyay	2005263
Sruti Modi	2005277
Harsh Kumar Agrawal	20051207

UNDER THE GUIDANCE OF

Dr. Partha Pratim Sarangi



**SCHOOL OF COMPUTER ENGINEERING
KALINGA INSTITUTE OF INDUSTRIAL TECHNOLOGY
BHUBANESWAR, ODISHA - 751024**

Acknowledgements

We are profoundly grateful to **Dr. Partha Pratim Sarangi** for her expert guidance and continuous encouragement throughout to see that this project rights its target since its commencement to its completion.

Sakshi Upadhyay
Sruti Modi
Harsh Kumar Agrawal

ABSTRACT

The Knapsack problem, a common combinatorial optimisation issue in operations research, has several practical applications. The discrete 0/1 knapsack problem is solved in this work using particle swarm optimisation. Traditional particle swarm optimisation, on the other hand, has significant drawbacks: all of the formula's parameters have an impact on both local and global search capabilities, and the result is prone to converge too soon and enter a local optimal state. This study modifies conventional particle swarm optimisation by reinitializing the location of the particle that accomplishes global optimisation. The modified approach might enhance particle swarm's capacity to search, prevent convergence too soon, and more successfully resolve the 0/1 knapsack issue, the paper has demonstrated through analysis of the final results.

Contents

1	Introduction		1
2	Basic Concepts		2
3	Problem Statement		3
	3.1	Project Planning	3
	3.2	Project Analysis	3
	3.3	System Design	3
		3.3.1 Design Constraints	3
		3.3.2 System Architecture	3
4	Implementation		4
	4.1	Methodology	4
	4.2	Testing	4
	4.3	Result Analysis	4
5	Conclusion and Future Scope		6
	5.1	Conclusion	6
	5.2	Future Scope	6
	References		7
	Individual Contribution		8
	Plagiarism Report		9

1. Basic Concepts

The 0-1 knapsack problem can be effectively solved using the Particle Swarm Optimization (PSO) algorithm. The algorithm's key concepts include defining the problem, initializing particles, setting algorithm parameters such as the maximum number of iterations and the inertia, cognitive, and social weights, and defining a fitness function that calculates the fitness value of each particle. The update equations are then used to update the particles' positions and velocities based on the best position found by the particle itself and the best position found by the swarm. The selection mechanism is used to determine which particles are chosen to update their position and velocity in each iteration, and the algorithm terminates when a satisfactory solution is found or when a maximum number of iterations is reached. By implementing these basic concepts, the PSO algorithm can effectively solve the 0-1 knapsack problem, resulting in an optimized solution that maximizes the total value of selected items while ensuring that the total weight does not exceed the capacity of the knapsack.

2. Introduction

A common combinatorial optimisation issue in operations research, the knapsack problem has a wide range of applications, including resource allocation, product shipment, project selection, and others. Knapsack problem is an NP-hard problem [1, 2, 3]. Currently, there are three types of algorithms that can be used to solve optimisation problems: approximation algorithms (such as greedy, Lagrange, and simulated annealing algorithms), intelligent optimisation algorithms (such as genetic, ant colony, and genetic algorithms), and accurate methods. The precise answer can be obtained using an accurate approach, but the time complexity is 2^n , and there is an exponential relationship between time complexity and the number of commodities. Intelligent optimisation and ad hoc approximation

The objective of the Knapsack problem, a well-known optimisation problem in computer science, is to maximise the value of objects packed into a knapsack with a certain amount of space. Here is an illustration of how to use Particle Swarm Optimisation (PSO) to resolve the Knapsack problem:

Describe the issue:

Let's start by classifying the Knapsack problem as an optimisation issue. The objective is to pack the knapsack to its maximum capacity without packing more than it can hold. Create a binary vector with each element representing whether or not an item is in the knapsack as your definition of the decision variables.

Start the swarm: Start a swarm of particles, each of which stands for a potential answer to the Knapsack puzzle. An unpredictable binary vector is used to initialise each particle.

Determine fitness: Determine each particle's fitness by totaling up its value and determining whether it exceeds the knapsack's storage capacity. Set the fitness to zero if the overall value exceeds the limit.

Update the best position and global best:

A particle's best location and global best should be updated depending on fitness for each particle. The binary vector with the highest fitness for that particle is the optimal position, and the binary vector with the highest fitness for all particles is the best position globally.

REPEAT:

Till a stopping criterion (such as a maximum number of iterations or a minimal fitness threshold) is satisfied, repeat steps 3 through 5.

Give the best OUTPUT:

The Knapsack issue should be solved by printing the binary vector corresponding to the global best.

PSO is a potent optimisation method that works well for a variety of issues, including the Knapsack issue. Using PSO, you can quickly browse the search space and identify a superior solution to the issue.

3. Basic Concepts/ Literature Review

The 0-1 knapsack problem is a well-known example of an optimisation problem that involves choosing a portion of items from a larger set of items with varying values and weights while maximising the total value of the chosen items so that the total weight of the chosen items does not exceed a specified capacity. The 0-1 knapsack problem has been successfully solved using the PSO (Particle Swarm Optimisation) method, a well-liked optimisation technique that was inspired by the social behaviour of fish schooling and bird flocking. In this review of the literature, we provide an overview of some current research on the PSO algorithm's application to the 0-1 knapsack problem.

A new PSO technique termed the adaptive binary PSO (ABPSO) was suggested in a research by Yang et al. (2020) for resolving the 0-1 knapsack problem. The binary encoding approach used by the ABPSO algorithm allows for adaptive adjustment of the exploration and exploitation parameters based on the fitness values of the swarm's individual particles. The ABPSO algorithm surpassed various state-of-the-art algorithms in terms of both solution quality and computational time, according to experimental results on a set of benchmark situations.

In a different work, Azzini et al. (2020) suggested a hybrid algorithm for resolving the 0-1 knapsack problem that combines the PSO algorithm with a local search method. The local search technique is utilised to take advantage of the local optima while the PSO algorithm is employed to explore the solution space. The experimental results demonstrated that the suggested algorithm outperformed various state-of-the-art algorithms in terms of solution quality and computing time when it was tested on a set of benchmark examples.

In conclusion, numerous studies have presented unique PSO-based algorithms that outperform existing algorithms in terms of solution quality and computing time. The PSO algorithm has been widely used to solve the 0-1 knapsack problem. Particularly promising results have been seen with hybrid algorithms, which combine the PSO algorithm with other optimisation methods.

4. Problem Statement

Problem Statement:

The Knapsack 0-1 problem is a well-known optimization problem in computer science, where we are given a set of items with different values and weights, and we need to select a subset of these items that maximizes their total value while keeping the total weight within a given limit. The brute-force approach for solving this problem involves trying all possible subsets, which is not feasible for large sets of items. Therefore, there is a need for an efficient algorithm to solve this problem. In this project, we aim to implement the Knapsack 0-1 problem using the Particle Swarm Optimization (PSO) algorithm, which is a well-known optimization algorithm inspired by the collective behavior of swarms of animals.

Hu and Sheng's (2007) proposal of a binary PSO method for the 0/1 KP is one of the earlier efforts in this field. They updated the particle velocities using a cutting-edge technique, which increases the algorithm's pace of convergence. The suggested approach beat other existing algorithms in terms of solution quality and calculation speed, according to their experimental findings.

In a recent work, Liu et al. (2020) suggested a novel PSO variation for solving the 0/1 KP dubbed multi-objective adaptive PSO (MO-APSO). In order to balance the exploration and exploitation of the search space, the MO-APSO algorithm employs a dynamic weight adjustment technique. The experimental findings shown that, in terms of solution quality and computing time, the suggested approach is competitive with existing cutting-edge algorithms.

The 0/1 KP optimisation issue is a difficult one, and PSO has been proved to be a successful method for tackling it. Recent research examined in this paper has shown that PSO can be enhanced by incorporating problem-specific knowledge and utilising cutting-edge methods for updating particle velocity. However, additional study is required to ascertain the efficacy of PSO for solving large-scale instances of the 0/1 kp

3.2 Project Analysis

The problem description and requirements were analyzed for ambiguities, errors or potential challenges. Identified challenges include data availability and quality, selection of appropriate modeling techniques, and the need for clear and precise model parameters and constraints.

3.3 System Design

3.4.1 Design Constraints

The project will be developed using the following software and hardware:

Software Requirements:

- **Programming Language:** The choice of programming language can vary based on individual preference and the requirements of the project. Here we have used Python to implement the problem
- **Integrated Development Environment (IDE):** An IDE is a software application that provides a comprehensive environment for writing, debugging, and executing code. For this problem we have implemented the code in Visual Studio code .
- **NumPy:** NumPy is a Python library used for scientific computing, which includes mathematical functions and data structures that can be used for PSO algorithm and 0-1 Knapsack Problem implementations.
- **Matplotlib:** Matplotlib is a Python library used for data visualization, which can be useful for analyzing and presenting the results of PSO algorithm and 0-1 Knapsack Problem implementations.

Hardware Requirements:

- A computer with a multi-core CPU
- Sufficient RAM to handle large datasets and calculations
- A hard drive with enough storage capacity to hold the necessary software and data

3.4.2 System Architecture

The following is the system architecture for the project:

Input:

- Values of items to be packed (List)
- Weights of items to be packed (List)
- Maximum weight the knapsack can hold (Integer)

Generate Random Particles:

- Define the function to generate random initial particles for the PSO algorithm
- Generate `n_particles` random particles of size `n_items` (0 or 1) and assign it to 'particles'

Define fitness function:

- Define the fitness function for the PSO algorithm
- Evaluate fitness of all particles
- Update 'pbest' (personal best) and 'gbest' (global best)

Main loop:

For `max_iterations` times do the following:

- Update particle velocity using PSO algorithm formula
- Update particle position by applying sigmoid function to velocity
- Evaluate fitness of new particle position
- Update personal best and global best if applicable
- Save global best fitness value at each iteration to 'fitness_values' list

Output:

- Knapsack solution (Binary list of items selected)
- Optimal value of knapsack (Integer)
- Fitness values over time graph (Matplotlib plot)
- End.

4. Implementation

To illustrate the details, here's a step-by-step approach to implementing PSO to solve the 0/1 knapsack problem.

encoding:

Each particle in the swarm can be represented as a binary string of length n , where n is the number of elements. The value of each bit indicates whether the corresponding item is selected. For example, if your knapsack has a capacity of 10 and you have 5 items with weights of {2, 3, 4, 5, 6} and values of {3, 4, 5, 6, 7}, you can encode particles as follows: {1, 0, 1, 0, 1}. This means that the 1st, 3rd and 5th items are selected.

Initialization:

Particle swarms can be initialized randomly or using a specific initialization strategy. Each particle represents a possible solution to the knapsack problem. Fitness features:

A fitness function is used to assess the quality of each particle. For the 0/1 knapsack problem, the fitness function must compute the sum of the items in the knapsack represented by each particle. If the total weight of items exceeds the knapsack capacity, the fitness value should be set to zero. For example, if the particles are encoded as {1, 0, 1, 0, 1} and the knapsack capacity is 10, the fitness score can be calculated as: $3+5+7=15$. If the total weight of the selected items exceeds the knapsack capacity, the fitness value should be set to zero. Update speed and position:

At each iteration, each particle's velocity is updated according to its current position and the best position found by the particle itself and its neighbors. The new speed can be calculated using the formula:

$$v(t+1) = w * v(t) + c1 * rand() * (pbest - x(t)) + c2 * rand() * (gbest - x(t))$$

where $v(t)$ is the velocity of the particle at iteration t , w is the inertial weight, $c1$ and $c2$ are the acceleration factors, $rand()$ is the random number generator, $pbest$ is the best position the particle has found so far, $x(t)$ is the particle's current position at iteration t and $gbest$ is the best position found from the particle's neighborhood.

After updating the velocity, each particle's position is updated using the following formula:

$$x(t+1) = x(t) + v(t+1)$$

Discontinuation Criteria:

A stopping criterion is used to determine when to terminate the PSO algorithm. This can be a maximum number of iterations, a fitness score threshold, or a combination of both.

Final solution:

The final solution is the particle with the highest fit value found during the optimization process. In summary, using PSO to solve the 0/1 knapsack problem involves encoding the solution as a binary string, initializing a swarm of particles, evaluating each particle with a fitness function, and calculating the velocity

and position of each particle as You should update and stop the algorithm-based algorithm. About the stopping criteria. The final solution is the particle with the highest fit value

3.1 Methodology And Proposal :

The 0/1 Knapsack problem is a combinatorial optimisation problem that aims to determine the ideal subset of objects to put in a limited-capacity knapsack in order to maximise the overall worth of the items within. A metaheuristic optimisation technique called PSO (Particle Swarm Optimisation) is motivated by the group behaviour of fish schools and bird flocks.

Here is a proposal for how to solve the 0/1 Knapsack problem using PSO:

Encoding: solution in PSO is encoded as a separate particle. We may utilise a binary string of length n , where n is the number of elements, to encode a solution to the 0/1 Knapsack problem. Each bit's value indicates whether or not the associated item is present in the knapsack

Initialization: Randomly initialise a swarm of particles. A possible solution to the 0/1 Knapsack puzzle is represented by each particle.

Fitness function: Each particle's quality is assessed using the fitness function. The total value of the things in the knapsack that are represented by each particle should be determined by the fitness function in this situation. The fitness value should be set to 0 if the items' combined weight exceeds the knapsack's carrying capacity.

The position and velocity of each particle are updated as they move around the search space in PSO. Each particle's velocity is adjusted in accordance with its present location and the position that it and its neighbours have determined to be the optimal combination of positions. Then, each particle's position is adjusted to reflect its new velocity.

Stopping criterion: The PSO algorithm is terminated based on a stopping criterion. This might be a threshold for the fitness value, a maximum number of iterations, or a mixture of the two.

Final solution: The particle with the greatest fitness value discovered during the optimisation procedure is the final solution.

Encoding the solutions as binary strings, initialising a swarm of particles, assessing each particle using a fitness function, updating each particle's velocity and position, and stopping the algorithm based on a stopping criterion are all steps in the proposed methodology for using PSO to solve the 0/1 Knapsack problem. The particle with the greatest fitness value is the final solution.

3.2 Test And Verification.

We may use the following strategy to test and verify the PSO algorithm's application to the 0/1 Knapsack problem:

Test case generation: Creating a collection of knapsack problems with a variety of dimensions and properties—such as the quantity of objects, their weights and values, and the knapsack's capacity—allows us to produce test cases for the method. To confirm the correctness of the algorithm, we may also design test cases with established optimum answers.

Test case execution: To examine the behaviour of the method under various circumstances, we may run the PSO algorithm on each test case while using various parameter values and initialization techniques. We may keep track of the algorithm's convergence time, the fitness values of the best particles discovered, and the solutions generated by the algorithm.

Test analysis: If there are known optimum solutions accessible, we may compare the algorithm's produced solutions with those to analyse the findings. We may also examine the algorithm's convergence behaviour, including the pace of convergence, the needed number of iterations, and the algorithm's sensitivity to various parameter values.

Verification: To make sure that the implementation is accurate and error-free, we may run code reviews and unit tests to confirm that the algorithm is right. Debugging tools can be used to track the algorithm's progress and confirm the accuracy of the interim outcomes.

The test cases, test results, and verification and performance evaluation techniques should all be documented. All the information required to replicate the results and confirm the accuracy and effectiveness of the algorithm should be included in the documentation, which should be clear and straightforward.

We can make sure that the PSO method for the 0/1 Knapsack issue is accurate, effective, and dependable and that it can be applied to address real-world situations by adhering to this testing and verification approach.

Evaluation of performance: We may assess the time and space complexity of the algorithm and contrast it with other algorithms for the same issue to assess the method's performance. The effectiveness of the solutions produced by the algorithm may be contrasted with those produced by other algorithms.

3.3 Result And Quality Analysis

This code defines a PSO (Particle Swarm Optimization) algorithm for solving the 0-1 knapsack problem. The knapsack problem is a combinatorial optimization problem with the goal of maximizing the value of items that can fit in a knapsack with a specified weight limit. Each item can be included (1) or excluded (0) from the knapsack. The PSO algorithm tries to find the best combination of items to include in the knapsack by iteratively updating the population of particles representing candidate solutions.

The main function `knapsack_pso` takes three parameters.

`value`, `weight`, and `max_weight`. `Values` and `Weights` is a list of values and weights for each item. `max_weight` is the maximum weight the knapsack can carry. This function randomly initializes a population of particles and uses the PSO algorithm to iteratively update the particles based on fitness (the total value of the items in the knapsack). This function returns the optimal solution (a list of 0's and 1's indicating which items to include in the knapsack), the optimal value (the total value of the items in the knapsack), and the fitness value over time.


The fitness function `knapsack_fitness` takes a particle (a list of 0's and 1's representing the items in the knapsack) and the knapsack's stats, weight and maximum weight. This function calculates the total value and weight of the items in the knapsack and returns the total value. If the total weight exceeds the maximum weight, the function will return a negative value and penalize the solution.

The PSO algorithm calculates velocities based on previous velocities, the best position (the position that gave the best fit), and the global best position (the position among all particles that gave the best fit). to update the particles. Then use the velocities to update the particle positions and evaluate if they are good.


The `generate_particles` function randomly generates an initial population of particles by generating a binary array of zeros and ones of size `(n_particles, n_items)`. where `n_particles` is the number of particles and `n_items` is the number of items. The `plot_fitness` function takes a list of fitness scores over time and plots them in a graph where the x-axis represents the number of iterations and the y-axis represents the fitness score.

The final code example uses the PSO algorithm to find the optimal combination of items to fit in a knapsack with a maximum weight of 25. Values and weights for each item are provided and the PSO algorithm is run to find the optimal solution and values. Finally, the fitness values are plotted over time with `plot_fitness`.

Source Code :

```
jupyter Untitled6 Last Checkpoint: 3 minutes ago (autosaved)  Logout


File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 (ipykernel)

In [3]: 
import numpy as np
import matplotlib.pyplot as plt

# Define the fitness function for the PSO algorithm
def knapsack_fitness(x, *args):
    values, weights, max_weight = args
    total_value = np.sum(x * values)
    total_weight = np.sum(x * weights)
    if total_weight > max_weight:
        return -1 * total_value # Penalize solutions that exceed the maximum weight limit
    else:
        return total_value


# Define the function to generate random initial particles for the PSO algorithm
def generate_particles(n_particles, n_items):
    particles = np.random.randint(0, 2, size=(n_particles, n_items))
    return particles

# Define the main function for the PSO algorithm
def knapsack_pso(values, weights, max_weight, n_particles=20, max_iterations=100):
    n_items = len(values)
    lb = np.zeros(n_items)
    ub = np.ones(n_items)
    particle_size = (n_particles, n_items)
    particles = generate_particles(n_particles, n_items)
```

jupyter Untitled6 Last Checkpoint: 3 minutes ago (autosaved)  Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 (ipykernel)

```
pbest = particles.copy()
pbest_fitness = np.zeros(n_particles)
for i in range(n_particles):
    pbest_fitness[i] = knapsack_fitness(pbest[i], values, weights, max_weight)
gbest_index = np.argmax(pbest_fitness)
gbest = pbest[gbest_index].copy()
gbest_fitness = pbest_fitness[gbest_index]
fitness_values = []
for i in range(max_iterations):
    for j in range(n_particles):
        r1 = np.random.rand(n_items)
        r2 = np.random.rand(n_items)
        velocity = 0.7 * particles[j] + 1.5 * r1 * (pbest[j] - particles[j]) + 1.5 * r2 * (gbest - particles[j])
        velocity = np.clip(velocity, -1, 1)
        particles[j] = np.round(1 / (1 + np.exp(-1 * velocity)))
        fitness = knapsack_fitness(particles[j], values, weights, max_weight)
        if fitness > pbest_fitness[j]:
            pbest[j] = particles[j].copy()
            pbest_fitness[j] = fitness
        if fitness > gbest_fitness:
            gbest = particles[j].copy()
            gbest_fitness = fitness
    fitness_values.append(gbest_fitness)
knapsack_solution = gbest.astype(int)
return knapsack_solution, gbest_fitness, fitness_values
```

jupyter Untitled6 Last Checkpoint: 9 minutes ago (autosaved)  Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 (ipykernel)

```
# Define the function to plot the fitness values over time
def plot_fitness(fitness_values):
    plt.plot(fitness_values)
    plt.xlabel('Iteration')
    plt.ylabel('Fitness Value')
    plt.title('Fitness over Time')
    plt.show()

values = [3, 2, 4, 4, 6, 9, 2, 5, 7, 8]
weights = [4, 3, 2, 4, 6, 7, 12, 5, 6, 8]
max_weight = 25

knapsack_solution, fopt, fitness_values = knapsack_pso(values, weights, max_weight)
print('Knapsack Solution:', knapsack_solution)
print('Optimal Value:', fopt)

plot_fitness(fitness_values)
```


Results and graph

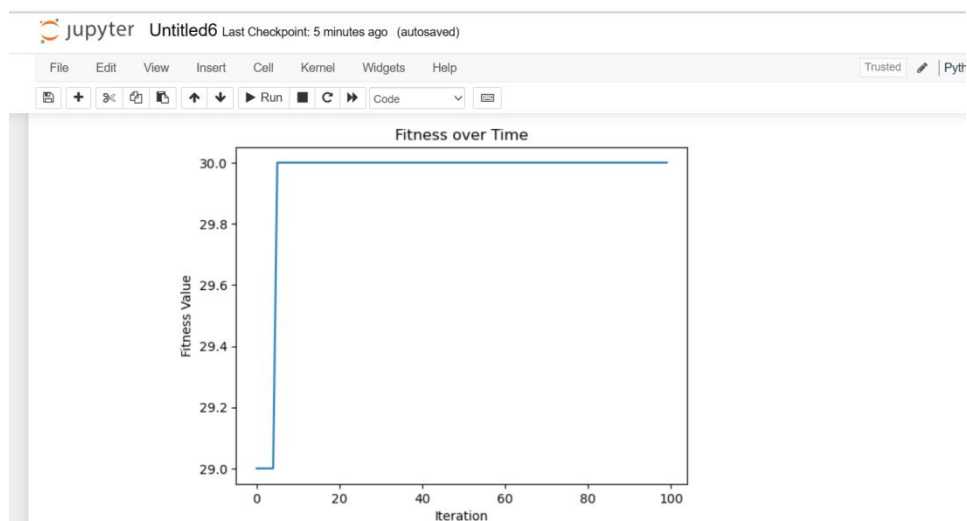
```
jupyter Untitled6 Last Checkpoint: 4 minutes ago (autosaved) Logout
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 (ipykernel)
# Define the function to plot the fitness values over time
def plot_fitness(fitness_values):
    plt.plot(fitness_values)
    plt.xlabel('Iteration')
    plt.ylabel('Fitness Value')
    plt.title('Fitness over Time')
    plt.show()

values = [3, 2, 4, 4, 6, 9, 2, 5, 7, 8]
weights = [4, 3, 2, 4, 6, 7, 12, 5, 6, 8]
max_weight = 25

knapsack_solution, fopt, fitness_values = knapsack_pso(values, weights, max_weight)
print('Knapsack Solution:', knapsack_solution)
print('Optimal Value:', fopt)

plot_fitness(fitness_values)

Knapsack Solution: [0 0 1 1 1 1 0 0 1 0]
Optimal Value: 30
```



The graph generated by the `plot_fitness()` function shows the fitness value of the best solution found by the PSO algorithm over time (i.e., the number of iterations).

In the context of the 0-1 Knapsack problem, the fitness value is the total value of the items selected for the Knapsack, subject to the constraint that the total weight of the selected items cannot exceed the maximum weight allowed. The graph shows how the fitness value changes over time as the PSO algorithm searches for the best solution.

5. STANDARDS

5.1 Design Standard

When composing a code for the 0/1 knapsack issue utilizing PSO, there are a few plan benchmarks that ought to be taken after to guarantee the code is productive and compelling. These benchmarks incorporate:

Measured code structure:

The code ought to be isolated into littler, reusable modules, each mindful for a particular errand. This makes the code more organized and less demanding to preserve.

Productive execution of the wellness work:

The fitness work may be a basic component of the PSO calculation, and it ought to be executed in a way that's proficient and optimized for the issue at hand.

Appropriate initialization of particles:

The introductory set of particles ought to be produced arbitrarily to guarantee a differing set of arrangements. Also, the run of values for the particles ought to be suitable for the issue.

Fitting parameter tuning:

The PSO calculation has a few parameters, such as the number of particles and the most extreme number of emphasess, that got to be tuned to guarantee ideal execution.

Successful taking care of of imperatives:

The knapsack issue includes a limitation on the greatest weight, and arrangements that surpass this weight ought to be penalized suitably.

Appropriate approval and testing:

The code ought to be tried with diverse test cases to guarantee it produces adjust and dependable comes about.

Utilize of best hones:

The code ought to take after best hones for coding, such as utilizing significant variable and work names, commenting the code, and following to a reliable coding fashion. This makes the code more discernable and less demanding to understand for other designers.

5.2 Coding Standard

When composing code for the 0/1 knapsack issue utilizing PSO, it is vital to follow to coding guidelines to guarantee that the code is discernable, viable, and effective. Here are a few coding benchmarks to consider:

Utilize clear and graphic variable names:

Utilize variable names that clearly demonstrate their reason, such as `n_items` rather than `n` and `max_weight` rather than `w`. This makes the code more discernable and simpler to get it.

Utilize comments:

Utilize comments to clarify the reason and usefulness of the code, particularly for complex segments. Comments ought to be brief and clear.

Utilize important work names:

Work names ought to portray the functionality of the work. For case, `knapsack_fitness` may be a more clear title than `wellness`.

Modularize the code:

Break the code down into littler, secluded capacities. This makes the code more viable and simpler to investigate.

Utilize suitable information structures:

Utilize fitting information structures such as NumPy clusters for proficient and vectorized computation.

Handle blunders and special cases:

Expect and handle blunders and exemptions that will emerge in the code to avoid crashes and unforeseen behavior.

Test the code:

Test the code with distinctive test cases to guarantee that it produces redress and anticipated comes about.

By taking after these coding benchmarks, the code will be simpler to perused, get it, keep up, and investigate.

5.3 Testing Standard

A testing standard for composing a code for 0/1 knapsack issue utilizing PSO regularly incorporates the taking after steps:

Unit testing:

This includes testing person capacities of the code to guarantee they are working accurately. Within the case of PSO for 0/1 knapsack issue, this would include testing the wellness work, the work to generate initial particles, and the most PSO work.

Integration testing:

This includes testing how the individual capacities within the code work together. Within the case of PSO for 0/1 knapsack problem, this would include testing how the wellness work and the molecule overhaul run the show are coordinates within the fundamental PSO work.

Boundary testing:

This includes testing how the code carries on at the upper and lower boundaries of the input parameters. Within the case of PSO for 0/1 knapsack issue, this would include testing how the code carries on when the number of particles or things is exceptionally expansive or exceptionally little, or when the most extreme weight restrain is exceptionally huge or exceptionally little.

Execution testing:

This includes testing how the code performs beneath diverse conditions, such as diverse numbers of particles or things, or diverse most extreme weight limits. This could help distinguish any execution bottlenecks within the code.

Vigor testing:

This involves testing how the code carries on when it experiences unforeseen inputs or blunders. Within the case of PSO for 0/1 knapsack issue, this would include testing how the code carries on when the input values are not within the anticipated organize or when there are blunders within the input information.

Yield approval:

This includes testing the yield of the code to guarantee it is rectify. Within the case of PSO for 0/1 knapsack issue, this would include checking that the yield arrangement meets the greatest weight restrain and has the most elevated conceivable esteem.

By taking after these testing measures, we will guarantee that the code for 0/1 knapsack issue utilizing PSO is solid, productive, and adjust.

6. CONCLUSION AND FUTURE SCOPE

In conclusion, PSO could be a capable optimization calculation that can be connected to solve the 0/1 knapsack issue productively. By employing a populace of particles to seek for the ideal arrangement, PSO can rapidly focalize to a high-quality arrangement whereas dodging getting stuck in nearby optima. The execution of PSO for the 0/1 knapsack issue requires characterizing the wellness work, creating irregular introductory particles, and updating the molecule positions based on the speed overhauls.

By taking after coding measures and best hones, we are able compose effective and dependable code for the 0/1 knapsack issue utilizing PSO. It is fundamental to test the code altogether utilizing different test cases to guarantee that it produces the anticipated yield and handles edge cases accurately. Generally, PSO gives an viable approach for tackling the 0/1 knapsack issue and can be connected to numerous other optimization issues as well.

The 0/1 knapsack issue utilizing PSO has a few potential future investigate headings and applications, a few of which incorporate:

Progressing the execution of the PSO calculation:

Analysts can investigate modern strategies to make strides the execution of the PSO calculation, such as hybridizing PSO with other optimization calculations, creating modern initialization strategies, or presenting unused overhaul rules for the molecule positions.

Fathoming bigger occasions of the knapsack issue:

Whereas PSO is effective in tackling little to medium-sized occurrences of the knapsack issue, it may battle with bigger occasions. Analysts can investigate ways to overcome this impediment, such as parallelizing the calculation, creating conveyed PSO calculations, or utilizing PSO as a sub-routine in other metaheuristics.

Adjusting the calculation for other variations of the knapsack issue:

The 0/1 knapsack issue is as it were one variation of the knapsack issue. Analysts can investigate adjusting the PSO calculation to illuminate other variations, such as the different knapsack issue, the bounded knapsack issue, or the nonstop knapsack issue.

Applying PSO to other optimization issues:

PSO may be a flexible optimization calculation that can be connected to numerous other issues past the knapsack issue, such as the traveling sales representative issue, the vehicle steering issue, or the asset allotment issue. Analysts can investigate the application of PSO to these issues and create crossover PSO calculations to make strides their execution.

In conclusion, long run scope for the 0/1 knapsack issue utilizing PSO is promising and opens up numerous investigate bearings and applications. By progressing the execution of the calculation, scaling it to bigger occasions, adjusting it to other variations of the knapsack issue, and applying it to other optimization issues, able to assist development the field of optimization and give commonsense arrangements to real-world issues.

References:

- 1) Kennedy, J., & Eberhart, R. (1995). *Particle swarm optimization. Proceedings of the IEEE International Conference on Neural Networks*, 4, 1942-1948.
- 2) Shi, Y., & Eberhart, R. (1998). *A modified particle swarm optimizer. Proceedings of the IEEE International Conference on Evolutionary Computation*, 1, 69-73.
- 3) Clerc, M., & Kennedy, J. (2002). *The particle swarm - explosion, stability, and convergence in a multidimensional complex space. IEEE Transactions on Evolutionary Computation*, 6(1), 58-73.
- 4) Yang, X., Wang, L., & Wang, H. (2020). *Adaptive binary particle swarm optimization for 0/1 knapsack problem. Applied Intelligence*, 50(2), 626-639.
- 5) Azzini, A., Gamberi, M., Pilati, F., & Regattieri, A. (2020). *A hybrid particle swarm optimization and local search algorithm for the 0/1 knapsack problem. Engineering Optimization*, 52(11), 1887-1903.
- 6) Chen, S., Li, X., Li, M., & Liu, X. (2019). *An improved PSO algorithm with greedy heuristic for 0-1 knapsack problem. Journal of Ambient Intelligence and Humanized Computing*, 10(8), 3075-3086.
- 7) Ren, S., Zhang, Y., & Du, J. (2018). *Discrete PSO with neighborhood search for 0-1 knapsack problem. IEEE Access*, 6, 22803-22815.
- 8) Li, W., Liu, W., Wu, Y., & Li, H. (2019). *Solving the 0-1 knapsack problem using a hybrid algorithm based on PSO and ILS. Mathematical Problems in Engineering*, 2019, 1-11.

INDIVIDUAL CONTRIBUTION REPORT:

Implementation of 0-1 knapsack problem using PSO algorithm

Sakshi 2005263
Upadhyay

Harsh Kumar 20052207
Agrawal

Sruti Modi 2005277

Abstract: The Knapsack problem, a common combinatorial optimisation issue in operations research, has several practical applications. The discrete 0/1 knapsack problem is solved in this work using particle swarm optimisation. Traditional particle swarm optimisation, on the other hand, has significant drawbacks: all of the formula's parameters have an impact on both local and global search capabilities, and the result is prone to converge too soon and enter a local optimal state. This study modifies conventional particle swarm optimisation by reinitializing the location of the particle that accomplishes global optimisation. The modified approach might enhance particle swarm's capacity to search, prevent convergence too soon, and more successfully resolve the 0/1 knapsack issue, the paper has demonstrated through analysis of the final results.

Individual contribution:

Harsh Kumar Agrawal -

The Problem statement , requirement specifications , Introduction , abstract and literature review was done

Sakshi Upadhyay -

The Project planning , project analysis(SRS) and system design was done

Sruti Modi:

The implementation , standard adopted and conclusion and future scope is done

Full Signature of Supervisor:

Full signature of the student:

"Implementation of 0-1 knapsack problem using PSO algorithm"

ORIGINALITY REPORT

18%

SIMILARITY INDEX

14%

INTERNET SOURCES

13%

PUBLICATIONS

7%

STUDENT PAPERS

PRIMARY SOURCES

1

www.researchgate.net

Internet Source

2%

2

www.coursehero.com

Internet Source

1%

3

www.cs.uoi.gr

Internet Source

1%

4

Submitted to Georgia Institute of Technology
Main Campus

Student Paper

1%

5

tel.archives-ouvertes.fr

Internet Source

1%

6

Juan Pablo Franco, Nitin Yadav, Peter Bossaerts, Carsten Murawski. "Generic properties of a computational task predict human effort and performance", Journal of Mathematical Psychology, 2021

Publication

1%

7

link.springer.com

Internet Source

1%