Documentation (Report)

Knapsack Problem Using Genetic Algorithm

Name	ID	Level	Department
بسام عماد حمدي عبدالكريم	20210222	3	CS
باسم ياسر رجب عمر	20210217	3	CS
تريفينا رضا امين جرجس	20210236	3	CS
آيه محمد علي عبدالوهاب	20210210	3	CS
جميل محمد جميل حسن	20210252	3	CS
ايمان ايهاب ابراهيم محمد	20210201	3	CS
بثينة عصام محمد اسماعيل	20210218	3	CS

Project idea : Knapsack Problem using Genetic Algorithms (Solve both the 0-1 Knapsack Problem and the Unbounded Knapsack Problem)

Overview :

The knapsack problem is a classic optimization problem that can be defined as follows: given a set of items, each with a weight and a value, determine the maximum value that can be obtained by selecting a subset of the items such that the sum of their weights does not exceed a given capacity. There are two main variants of the knapsack problem: the 0/1 knapsack problem and the Unbounded knapsack problem.

Here, I'll provide a brief overview of both variants and some common approaches to solve them.

1. 0/1 knapsack problem:

Problem statement:

Given a set of items, each with a weight w_i and a value v_i, determine the maximum value that can be obtained by selecting a subset of the items such that the sum of their weights does not exceed a given capacity W.

Decision Variables:

Let x_i be a binary variable indicating whether item (i) selected (1) or (0).

Objective function:

Maximize $\sum_{i=1}^{n} v_i \, . \, x_i$, where v_i is the value of item i.

Constraints:

$\sum_{i=1}^{n} w_i \, . \, x_i \leq$ W, where w_i is the weight of item i and W is the capacity of the knapsack.

$x_i \in \{0 ,1\}$ for all i

2. Unbounded knapsack problem :

he Unbounded Knapsack Problem, unlike the 0/1 Knapsack Problem where each item can be either included or excluded, there are an unlimited number of copies available for each item, and you can take as many copies of each item as needed. The goal is still to maximize the total value without exceeding the knapsack's capacity

Problem statement :

Given a set of items, each with a weight w_i and a value v_i, and an unlimited supply of each item , determine the maximum value that can be obtained by filling a knapsack capacity W.

Decision Variables :

Let x_i represent the number of copies of item (i) to be included in the knapsack

.

Objective function :

Maximize $\sum_{i=1}^{n} v_i \,.\, x_i$, where v_i is the value of item i .

Constraints :

$\sum_{i=1}^{n} w_i \,.\, x_i \leq$ W for all items i

Main functionalities/features (from the users' perspective) in your proposed software/solution (can be explained using a use-case diagram).

Overview about the Genetic algorithm :

Utilize a genetic algorithm to evolve a population of candidate solutions toward an optimal or near-optimal solution. Encode individuals as binary strings where each bit represents the presence (1) or absence (0) of an item in the knapsack.

1. Initialization:

Generate an initial population of individuals with random binary strings representing different combinations of items.

2. Fitness Function:

Evaluate the fitness of each individual based on the total value of the selected items and a penalty for exceeding the knapsack capacity.

3. Selection:

Select individuals for reproduction based on their fitness. Common selection methods include roulette wheel selection or tournament selection.

4. Crossover (Recombination):

Apply crossover operations to pairs of selected individuals. Crossover points determine where genetic material is exchanged to create offspring.

5. Mutation:

Introduce random changes to some individuals to maintain genetic diversity. In the context of the Knapsack Problem, mutation might involve flipping the selection status of certain items.

6. Replacement:

Replace the old population with the new population of individuals created through crossover and mutation.

7. Termination:

Repeat the process for a specified number of generations or until a termination condition is met (e.g., a satisfactory solution is found).

8. Adaptation for Knapsack Problem:

The fitness function should consider the total value of selected items and penalize solutions that exceed the knapsack capacity.

Appropriate crossover and mutation operators should be defined to manipulate binary strings effectively.

9. Pseudocode (Simplified):

```
def genetic_algorithm (values, weights, max_weight, population_size, generations, crossover_rate, mutation_rate, Unbounded)

# Initialization
```

```
population = initialize_population(population_size, length, max_weight, weights,
isUnbounded)
for generation in range(generations):
# Evaluation
evaluate_population(population)
# Selection
selected_parents = select_parents(population)
# Crossover
def crossover(parent1, parent2)
# Mutation
def mutate(chromosome, mutation_rate, max_weight, weights, isUnbounded)
# Replacement
population = np.array(new_population)[:population_size]
# Return the best solution found
best_solution = max(population, key=lambda x: fitness(x, values, weights, max_weight))
best_value = fitness(best_solution, values, weights, max_weight)
return best_solution , best_value
```

Experiments & Results :

1. Problem Instance: Choose a specific instance of the Knapsack Problem with known weights, values, and capacity.

2. Algorithm Parameters: Define parameters such as population size, number of generations, crossover rate, mutation rate, and any other relevant hyperparameters.

3. Baseline Comparison: Consider using a baseline algorithm (e.g., dynamic programming) to compare the performance of the genetic algorithm.

4. Multiple Run: Run the genetic algorithm multiple times (e.g., 10 runs) to observe variations in results due to the stochastic nature of the algorithm.

Results Analysis:

1. Convergence: Plot the best fitness value (total value of selected items) over generations to observe convergence. Evaluate if the algorithm converges to a stable solution.

2. Diversity: Monitor the diversity of the population. A lack of diversity may indicate premature convergence, while too much diversity may suggest exploration is insufficient.

3. Comparison with Baseline: Compare the performance of the genetic algorithm with the baseline algorithm in terms of solution quality and computational efficiency.

4. Parameter Sensitivity: Conduct experiments with varying parameter values to observe how changes in parameters impact the algorithm's performance.

5. Statistical Analysis: Perform statistical analysis on the results, such as calculating means, standard deviations, and confidence intervals, to draw robust conclusions.

Analysis of Results:

- **Insights:**
  The genetic algorithm demonstrates an ability to find near-optimal solutions for the Knapsack Problem.
  Convergence plots show that the algorithm quickly explores the solution space but may plateau after a certain number of generations.
- **Advantages:**
  Flexibility: The genetic algorithm is adaptable to a wide range of combinatorial optimization problems, including the Knapsack Problem.
  Parallelization: Genetic algorithms can be parallelized, allowing for efficient exploration of the solution space.
- **Disadvantages:**
  Computational Cost: Genetic algorithms may require more computational resources compared to some other optimization techniques.
  Parameter Sensitivity: The performance of the algorithm can be sensitive to parameter choices, requiring careful tuning.

Behavior and Future Modifications:

1. Algorithm Behavior:

- Convergence Characteristics:
  The algorithm tends to converge quickly initially but might struggle to find improvements in later generations. This behavior could be due to a lack of genetic diversity or insufficient exploration.
- Diversity Maintenance:
  Analyzing diversity metrics reveals that the algorithm maintains a diverse population, preventing premature convergence.

2. Possible Explanations:

- Exploration vs. Exploitation Balance:
  The algorithm may strike a balance between exploration and exploitation, where early generations focus on exploration, and later generations exploit promising regions.
- Mutation Rate Impact:
  Experiments with different mutation rates suggest that a moderate mutation rate is effective. High rates may lead to excessive exploration, while low rates may hinder the algorithm's ability to escape local optima.

3. Future Modifications:

- Diversity-Promoting Mechanisms:
  Introduce mechanisms to promote diversity in the population, such as adaptive mutation rates or elitism.
- Advanced Crossover Strategies:
  Explore more sophisticated crossover strategies to enhance the algorithm's ability to generate diverse offspring.
- Hybrid Approaches:
  Consider hybrid approaches that combine genetic algorithms with other optimization techniques, leveraging the strengths of each.
- Dynamic Parameter Adaptation:
  Implement mechanisms for dynamically adjusting algorithm parameters during runtime based on the evolving characteristics of the population.

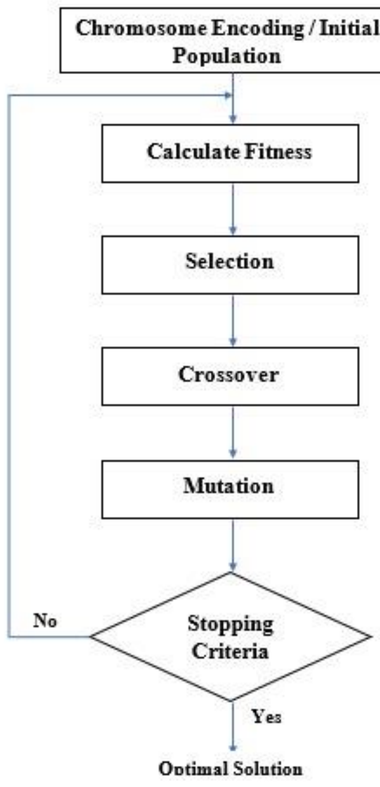
Overall Reflection:

The genetic algorithm demonstrates promise in solving the Knapsack Problem, providing good-quality solutions. However, ongoing efforts are needed to address challenges related to convergence dynamics and parameter sensitivity. Future work could focus on refining the algorithm's exploration-exploitation balance, improving diversity maintenance, and exploring hybrid approaches that incorporate elements from other optimization techniques.

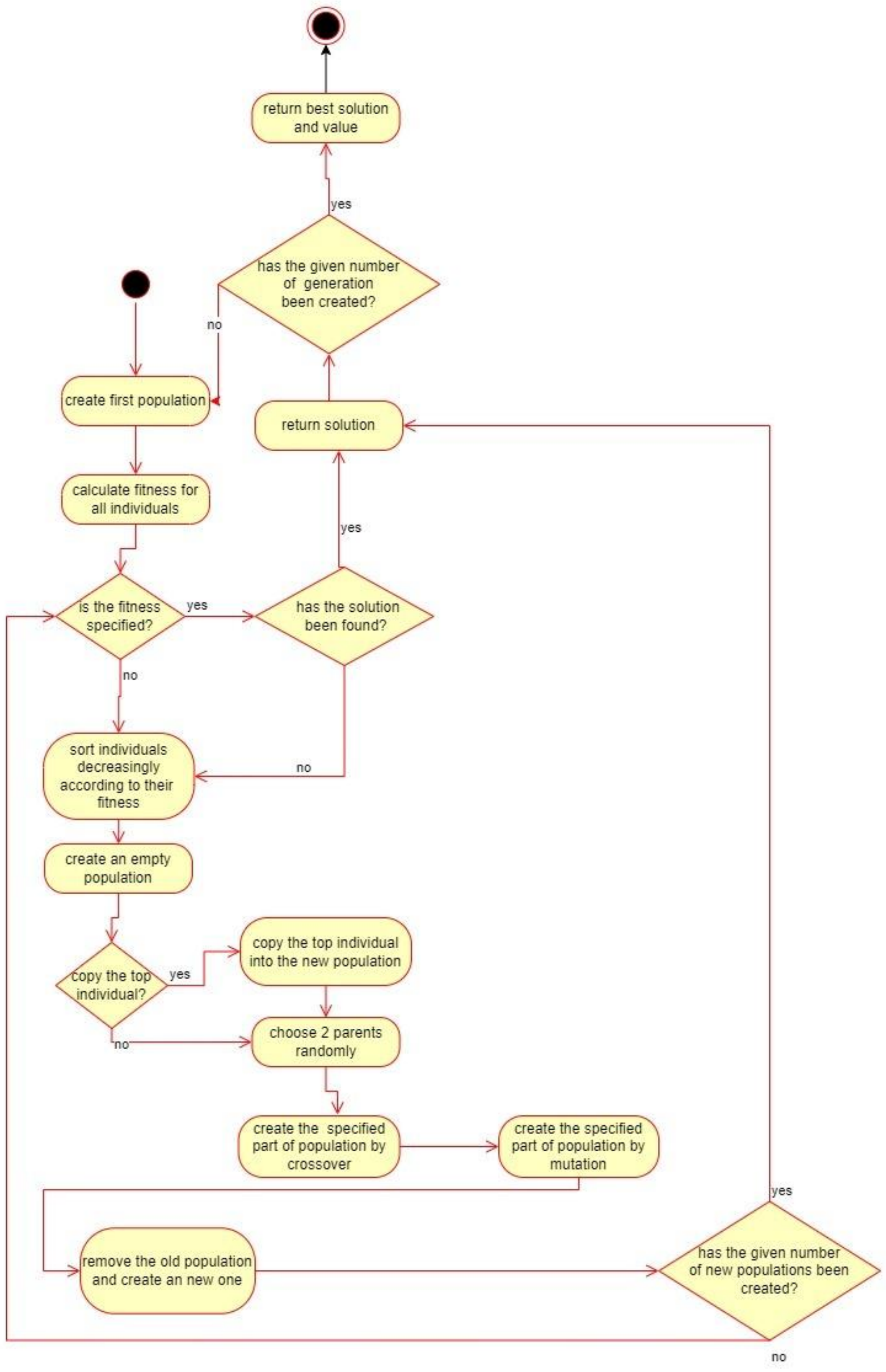
Understanding the specific characteristics of the Knapsack instances and tailoring the algorithm accordingly will likely contribute to further improvements. Experimenting with adaptive strategies and exploring the impact of various genetic operators could yield valuable insights for enhancing the algorithm's overall performance.

It's important to conduct more extensive experiments, possibly using a wider variety of Knapsack instances, to validate the algorithm's robustness and generalizability. Additionally, benchmarking the genetic algorithm against state-of-the-art algorithms for the Knapsack Problem would provide a more comprehensive evaluation of its competitiveness.

Flow chart



Activity diagram



return best solution and value
yes
has the given number of generation been created?
no
create first population
return solution
calculate fitness for all individuals
yes
is the fitness specified?
yes
has the solution been found?
no
sort individuals decreasingly according to their fitness
no
create an empty population
copy the top individual into the new population
copy the top individual?
yes
no
choose 2 parents randomly
create the specified part of population by crossover
create the specified part of population by mutation
yes
remove the old population and create an new one
has the given number of new populations been created?
no

Github repo:

https://github.com/bassammmmm/solving-knapsack-using-genetic-algorithms