Text

Description automatically generated

**Activity based**

**Project Report on**

**Airtificial Inteligence**

**Project Module - III**

**Submitted to Vishwakarma University, Pune**

**Under the Initiative of**

**Contemporary Curriculum, Pedagogy, and Practice (C2P2)**

**By**

**Maaz Sameer Shaikh**

**SRN No : 202101332**

**Roll No : 29**

**Div : E**

**Third Year Engineering**

**Department of Computer Engineering**

**Faculty of Science and Technology**

**Faculty Incharge:- Prof. Richa Agnihotri**

**Date Of Project 3:- 28/4/24**

**Academic Year**

**2023-2024**

**Software Engineering : Project Module III**

**Project Name : Solving Knapsack Problems using Hill Climbing**

**Problem Statement:**

The problem at hand is the Knapsack Problem Solver, which aims to efficiently solve the knapsack problem using different optimization algorithms. The knapsack problem involves selecting a combination of items, each with its own value and weight, to maximize the total value while keeping the total weight within a certain limit.

**Base Algorithm Used:**

The project employs a variation of the Hill Climbing algorithm to solve the Knapsack Problem. Hill Climbing is a local search algorithm that iteratively improves a candidate solution by making incremental changes. In this context, it attempts to find a solution that maximizes the total value while adhering to the weight constraints of the knapsack.

**Objectives:**

The objective appears to be creating a GUI application to solve the Knapsack problem, where users can input the number of items, maximum weight of the Knapsack, and maximum number of iterations for the algorithm. The solution should be displayed iteratively or as a final result.

**Methodology, Approaches, Workflow Diagram:**

**Methodology**

* **GUI Creation with tkinter:**
  + The tkinter library is used to create a graphical user interface (GUI) for the Knapsack problem solver.
  + The GUI includes input fields, buttons, and a scrolled text widget for displaying the solution.
* **Input Fields:**
  + Users are provided with input fields to specify:
  + Number of items: Users can input the total number of items to consider for the Knapsack problem.
  + Maximum weight of the Knapsack: Users specify the maximum weight capacity of the Knapsack.
  + Maximum number of iterations: Users define the maximum number of iterations for the randomized algorithm.
* **Buttons:**
  + "Add Items" Button: This button allows users to input the values and weights for each item.
  + "Solve Knapsack" Button: Initiates the solving process for the Knapsack problem based on the provided inputs.
* **Scrolled Text Widget:**
  + A scrolled text widget is utilized to display the solution.
  + It provides a scrollable area where the solution can be displayed iteratively or as a final result.

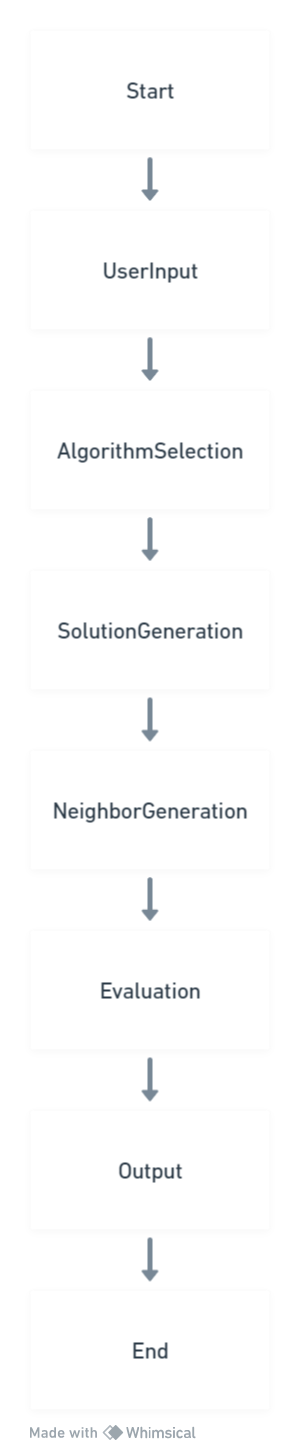
**Approaches:**

* **Adding Items:**
  + Users can input the value and weight for each item through the GUI.
  + For each item, an entry field is provided to input its value and weight.
* **Solving Knapsack:**
  + Initialization:

An initial random solution is generated where each item is either included or not included in the Knapsack.

* **Iterative Improvement:**
  + The algorithm iteratively tries to improve the solution.
  + For each iteration:
  + A neighbor solution is generated by randomly flipping the inclusion status of an item.
  + If the neighbor solution:
  + Is feasible (i.e., its weight does not exceed the maximum Knapsack weight), and
  + Improves the objective value (total value of items included),
  + Then it's accepted as the new current solution.
* **Termination:**
  + The iterations continue until either:
  + The maximum number of iterations specified by the user is reached, or
  + No further improvement is possible (local optimum reached).

**Workflow Diagram:**



**Code:**

import tkinter as tk

from tkinter import scrolledtext

import random

class KnapsackSolverGUI:

def \_\_init\_\_(self):

self.root = tk.Tk()

self.root.title("Knapsack Problem Solver")

self.root.geometry("600x500") # Set window size

self.root.configure(bg="#f0f0f0") # Set background color

# Create frame for input fields

input\_frame = tk.Frame(self.root, bg="#f0f0f0", bd=2, relief=tk.GROOVE)

input\_frame.pack(pady=20)

# Number of items entry

self.num\_items\_entry = tk.Entry(input\_frame, width=10, font=("Arial", 12), bd=1)

self.num\_items\_entry.grid(row=0, column=1, padx=10, pady=10)

self.num\_items\_label = tk.Label(input\_frame, text="Number of Items:", font=("Arial", 12), bg="#f0f0f0")

self.num\_items\_label.grid(row=0, column=0, padx=10, pady=10)

# Maximum weight entry

self.max\_weight\_entry = tk.Entry(input\_frame, width=10, font=("Arial", 12), bd=1)

self.max\_weight\_entry.grid(row=1, column=1, padx=10, pady=10)

self.max\_weight\_label = tk.Label(input\_frame, text="Maximum Weight of the Knapsack:", font=("Arial", 12), bg="#f0f0f0")

self.max\_weight\_label.grid(row=1, column=0, padx=10, pady=10)

# Maximum iterations entry

self.max\_iterations\_entry = tk.Entry(input\_frame, width=10, font=("Arial", 12), bd=1)

self.max\_iterations\_entry.grid(row=2, column=1, padx=10, pady=10)

self.max\_iterations\_label = tk.Label(input\_frame, text="Maximum Number of Iterations:", font=("Arial", 12), bg="#f0f0f0")

self.max\_iterations\_label.grid(row=2, column=0, padx=10, pady=10)

# Add items button

self.add\_items\_button = tk.Button(input\_frame, text="Add Items", font=("Arial", 12), bg="#4caf50", fg="white", relief=tk.FLAT, command=self.add\_items)

self.add\_items\_button.grid(row=3, column=0, columnspan=2, pady=10)

# Solve Knapsack button

self.solve\_button = tk.Button(input\_frame, text="Solve Knapsack", font=("Arial", 12), bg="#2196f3", fg="white", relief=tk.FLAT, command=self.solve\_knapsack)

self.solve\_button.grid(row=4, column=0, columnspan=2, pady=10)

# Create frame for item entries

self.items\_frame = tk.Frame(self.root, bg="#f0f0f0", bd=2, relief=tk.GROOVE)

self.items\_frame.pack(pady=20)

# Scrolled text widget to display solution

self.solution\_text = scrolledtext.ScrolledText(self.root, width=50, height=15, font=("Arial", 12), bg="#ffffff", bd=2, relief=tk.GROOVE)

self.solution\_text.pack(pady=20, fill=tk.BOTH, expand=True)

self.root.mainloop()

def add\_items(self):

try:

self.num\_items = int(self.num\_items\_entry.get())

self.max\_weight = float(self.max\_weight\_entry.get())

self.max\_iterations = int(self.max\_iterations\_entry.get())

# Clear previous item entries

for widget in self.items\_frame.winfo\_children():

widget.destroy()

# Entry fields for item values and weights

self.items = []

for i in range(self.num\_items):

value\_entry = tk.Entry(self.items\_frame, width=10, font=("Arial", 12), bd=1)

value\_entry.grid(row=i, column=1, padx=10, pady=5)

value\_label = tk.Label(self.items\_frame, text=f"Value for Item {i + 1}:", font=("Arial", 12), bg="#f0f0f0")

value\_label.grid(row=i, column=0, padx=10, pady=5)

weight\_entry = tk.Entry(self.items\_frame, width=10, font=("Arial", 12), bd=1)

weight\_entry.grid(row=i, column=3, padx=10, pady=5)

weight\_label = tk.Label(self.items\_frame, text=f"Weight for Item {i + 1}:", font=("Arial", 12), bg="#f0f0f0")

weight\_label.grid(row=i, column=2, padx=10, pady=5)

self.items.append((value\_entry, weight\_entry))

except ValueError as ve:

print(f"Error: {ve}")

def solve\_knapsack(self):

try:

items = [(float(value\_entry.get()), float(weight\_entry.get())) for value\_entry, weight\_entry in self.items]

current\_solution = self.generate\_random\_solution(items)

current\_value, current\_weight = self.calculate\_value(current\_solution, items)

solution\_history = [(current\_solution, current\_value, current\_weight)]

for i in range(1, self.max\_iterations + 1): # Start from 1 since we already have the initial solution

neighbor = self.generate\_neighbor(current\_solution)

neighbor\_value, neighbor\_weight = self.calculate\_value(neighbor, items)

if neighbor\_weight <= self.max\_weight and neighbor\_value > current\_value:

current\_solution = neighbor

current\_value = neighbor\_value

current\_weight = neighbor\_weight

solution\_history.append((current\_solution, current\_value, current\_weight))

else:

break # Stop iterating if there's no improvement

if solution\_history:

self.display\_solution("Final", current\_solution, current\_value, current\_weight) # Display the final solution

except ValueError as ve:

print(f"Error: {ve}")

def generate\_random\_solution(self, items):

return [random.randint(0, 1) for \_ in range(len(items))]

def calculate\_value(self, solution, items):

total\_value = sum(float(item[0].get()) for item, selected in zip(self.items, solution) if selected)

total\_weight = sum(float(item[1].get()) for item, selected in zip(self.items, solution) if selected)

return total\_value, total\_weight

def generate\_neighbor(self, current\_solution):

index = random.randint(0, len(current\_solution) - 1)

neighbor = current\_solution[:]

neighbor[index] = 1 - neighbor[index] # Flip 0 to 1 or 1 to 0

return neighbor

def display\_solution(self, iteration, solution, value, weight):

if iteration == "Final":

self.solution\_text.delete('1.0', tk.END) # Clear previous output

self.solution\_text.insert(tk.END, f"Final Solution:\n", "bold")

self.solution\_text.insert(tk.END, f"Solution: {solution}\n")

self.solution\_text.insert(tk.END, f"Total Value: {value}\n")

self.solution\_text.insert(tk.END, f"Total Weight: {weight}\n")

else:

self.solution\_text.insert(tk.END, f"Iteration {iteration}:\n", "bold")

self.solution\_text.insert(tk.END, f"Solution: {solution}\n")

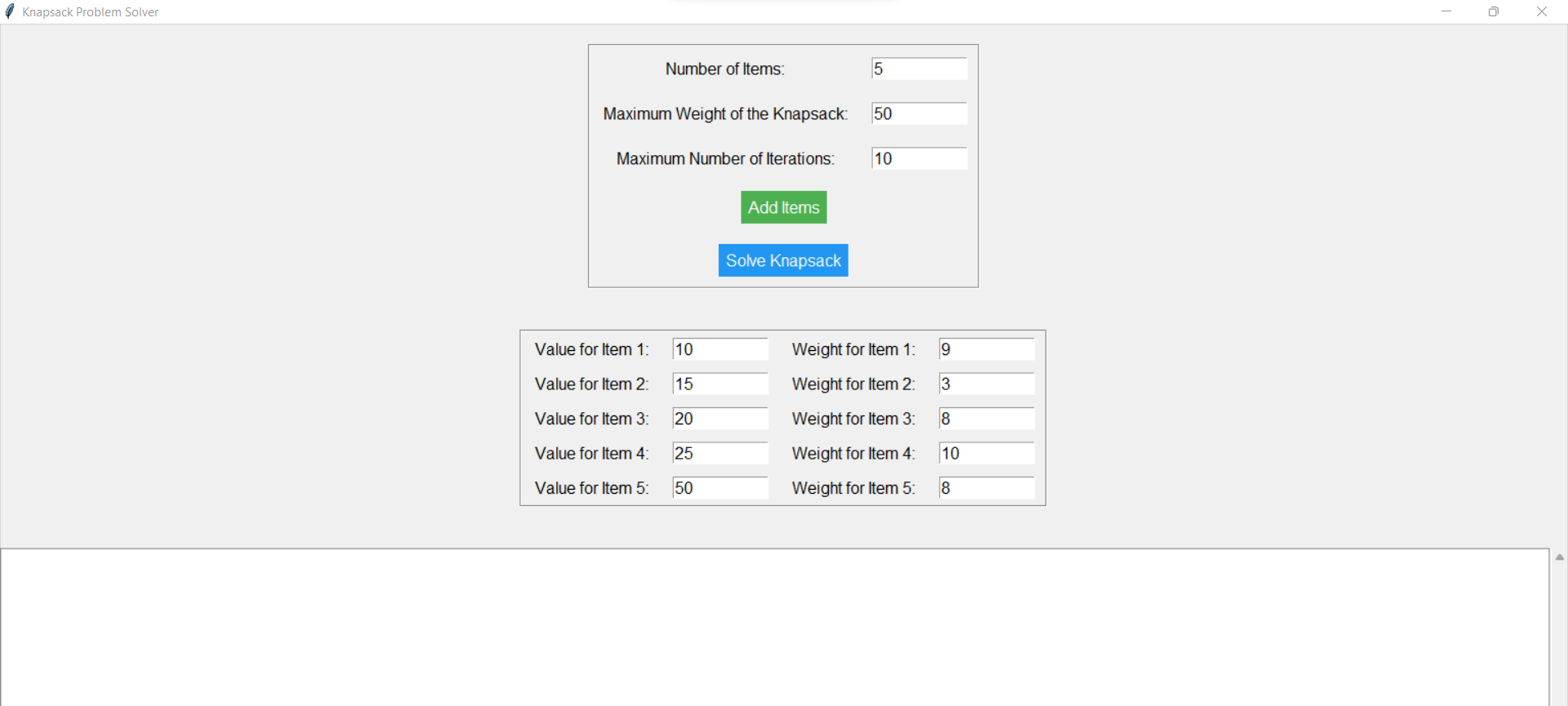
self.solution\_text.insert(tk.END, f"Total Value: {value}\n")

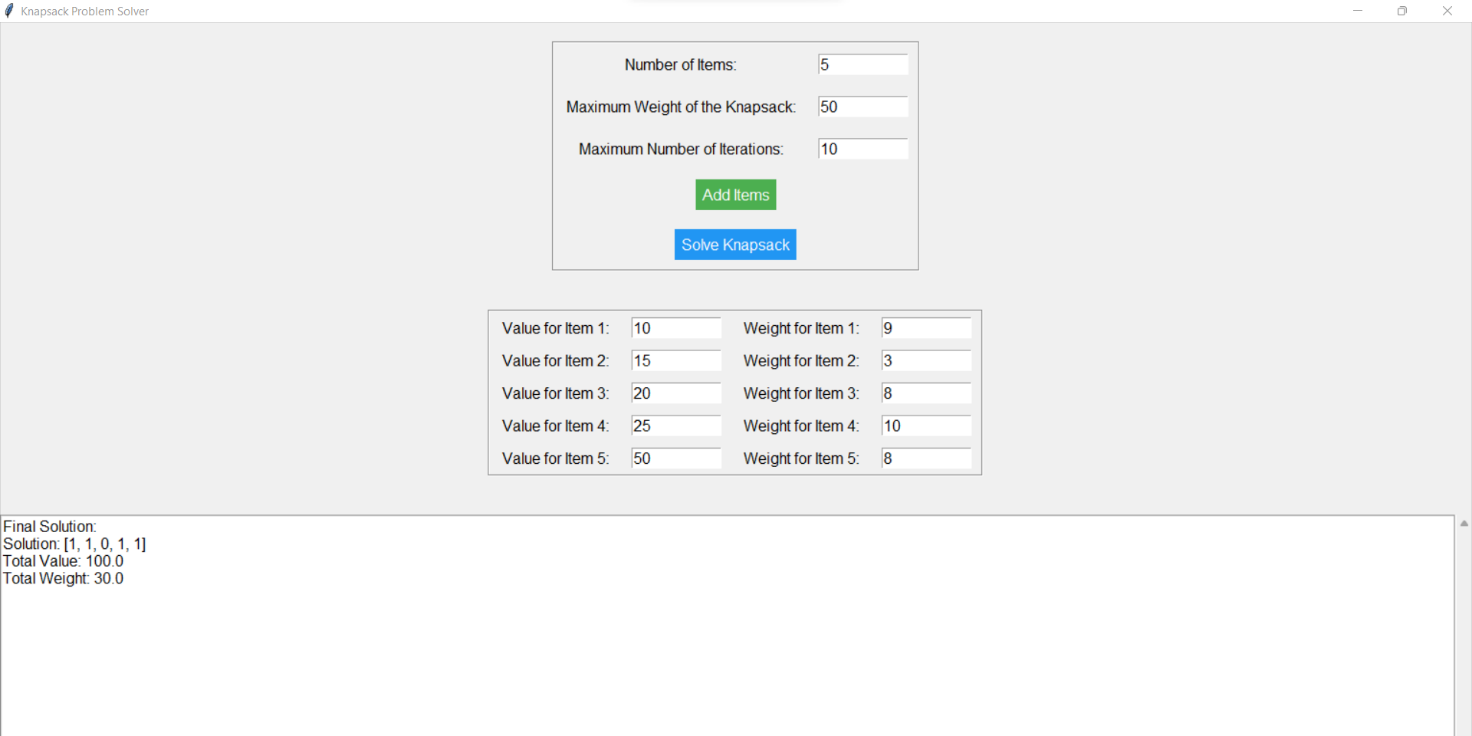
self.solution\_text.insert(tk.END, f"Total Weight: {weight}\n")

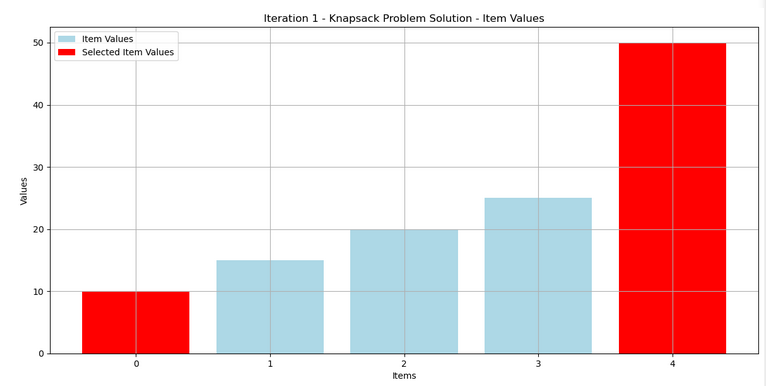
self.solution\_text.insert(tk.END, "\n", "normal")

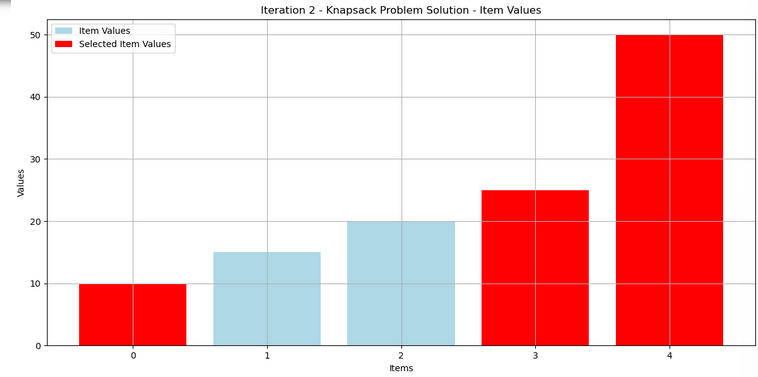
KnapsackSolverGUI()

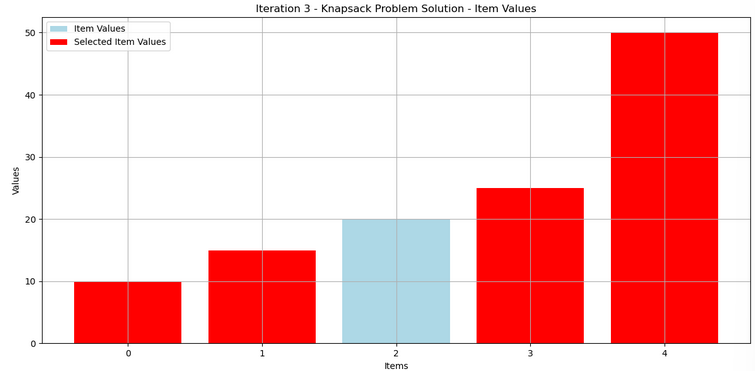
**Output:**

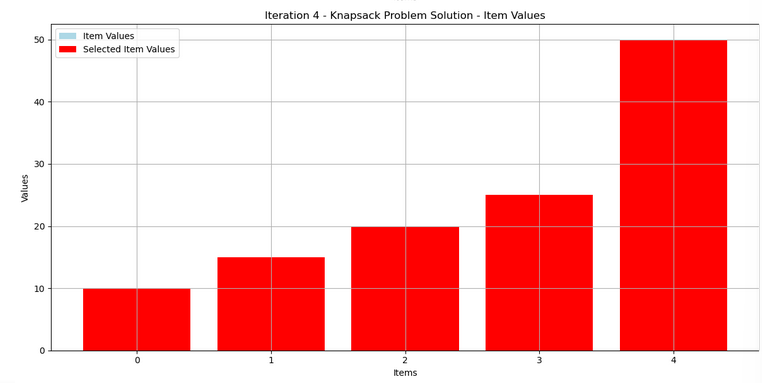
****

****









**Pseudo code or Algorithm:**

initialize GUI

create input fields for number of items, maximum weight, and maximum iterations

create buttons to add items and solve Knapsack

create scrolled text widget to display solution

function add\_items():

get input for number of items, maximum weight, and maximum iterations

clear previous item entries

create entry fields for item values and weights

store items in a list

function solve\_knapsack():

get items from input fields

generate initial random solution

calculate value and weight of initial solution

initialize solution history with initial solution

iterate until maximum iterations:

generate neighbor solution by flipping a random bit

if neighbor solution is feasible and improves value:

update current solution, value, and weight

add current solution to solution history

else:

break

display final solution

function generate\_random\_solution():

generate a list of random 0s and 1s of length equal to number of items

function calculate\_value():

calculate total value and weight of a solution

function generate\_neighbor():

generate a neighbor solution by flipping a random bit in the current solution

function display\_solution():

display solution in the scrolled text widget

**Conclusion:**

The code provides a functional GUI for solving the Knapsack problem using a randomized algorithm. Users can input item details and solve the problem interactively. However, improvements such as error handling and user feedback could enhance the usability of the application. Additionally, considering different algorithms for solving the Knapsack problem could be beneficial depending on the problem size and constraints.