PROJECT REPORT
Summer Intern TATA STEEL

Name-TANISHA BASU
Guided By-Saibal Nandi Sir
Mentor-Harsh Pateriya Sir

PURPOSE-

The Report describes various approaches that are tried to solve the 1D Cutting Stock Rebar Optimisation problem in order to get the demands fullfilled and minimise wastage

Contents-

1.The various steps to Knapsack and Linear programmming method along with greedy approach to distribute the demands optimally.

2.Tried Genetic approach to get the distribution optimally.

PROBLEM STATEMENT-

The 1D Cutting Stock Problem is an example of Optimisation problem and is used in manufacturing industries.Each rods have specific demands of specific lengths along with the arbitary(inventory values) of demands .The goal is to determine the cutting patterns by various approaches and determine the demands upto which it can be fulfilled and which inventory rods to be used first and in what manner.

Key Objectives-

1.Fulfill orders

2.Determine cutting patterns for fulfilling demands and reducing wastage

Approach-

1.Greedy Approach-Initially the greedy approach is being used to distribute the demands accordingly how it can be done

2.Linear Programming and Knapsack Technique-Utilise the Linear Programming and Knapsack Techniques to get the optimal cutting patterns that helps to fulfill the demands .

3.Tried Genetic Algorithm-Initially tried the genetic algorithm to distribute the demands accordingly.

4.Cutting Patterns-Generate the cutting patterns to fulfill the demands accordingly.

Benifits and need-

1.Cost Reduction- Optimizing the use of rebar can lead to significant cost savings by minimizing waste and reducing the amount of material needed.

2.Material Efficiency- Optimized rebar design and placement reduce the amount of steel required for construction, leading to more efficient use of materials

- INSIGHTS FROM THE METHODS-

1.Initial Distribution of the demands according to Greedy Approach-

"w": [1.4, 0.7, 3.41, 3.98, 0.765, 1.13,6.18],

"d": [96, 20340, 1278, 852, 4970, 3976,2520],

"W": [6.0, 6.5, 7.0, 7.5, 8.0, 8.5, 9.0, 9.5, 10.0,10.5]

Code snippet-

The code snippet distributes the demand greedily distributing the highest demand to highest width optimally.

```python
def distribute_demand(d, W):
    total_W = sum(W)
    distribution = [(W_i / total_W) * d for W_i in W]
    rounded_distribution = [int(round(dist)) for dist in distribution]
    adjustment = d - sum(rounded_distribution)

    if adjustment != 0:
        fractional_parts = [(distribution[i] % 1, i) for i in range(len(W))]
        fractional_parts.sort(reverse=True, key=lambda x: x[0])

        for _, idx in fractional_parts[:abs(adjustment)]:
            rounded_distribution[idx] += 1 if adjustment > 0 else -1

    return rounded_distribution
def get_formatted_results(data):
    formatted_results = []

    for i, w in enumerate(data['w']):
        d_value = data['d'][i]
        distribution = distribute_demand(d_value, data['W'])

        for j, W in enumerate(data['W']):
            if distribution[j] > 0:
                formatted_results.append(f"{w}\t{distribution[j]}\t{W}")

    return formatted_results
results = get_formatted_results(data)
for result in results:
    print(result)
```

On combining all the matrices and calculating we can get the demands fullfilled by width as follows-
(Greedy Approach)
#For w=1.4
fulfilled=96/96

#For w=0.7
fulfilled=20340/20340

#For w=3.41
fulfilled=1278/1278
[
#For w=3.98
fulfilled=852/852

#For w=0.765
fulfilled=4970/4970

#For w=1.13
fulfilled=3976/3976

#For w=6.18
fulfilled=2515/2520 [on calculating 99.7% effective]

```python
import numpy as np
def greedy_knapsack(W, w, d):
    w = np.array(w)
    d = np.array(d)

    n = len(w)
    total_weight = 0
    total_value = 0
    x = np.zeros(n)
    for i in range(n):
        if total_weight + w[i] <= W:
            x[i] = 1
            total_weight += w[i]
            total_value += d[i]
        else:
            fraction = (W - total_weight) / w[i]
            x[i] = fraction
            total_weight += w[i] * fraction
            total_value += d[i] * fraction
            break

    return x, total_weight, total_value

def generate_greedy_pattern(W, w, d):
    w = np.array(w)
    d = np.array(d)
    x, total_weight, total_value = greedy_knapsack(W, w, d)
    print("Weight vector (w):", w)
    print("Demand vector (d):", d)
```

Individually iteration of each width and sol.x to get the matrix row by row -

(Knapsack Technique Linear Programming)

Code snippet-The knapsack and linear programming approach helps to calculate the matrix and sol.x individual iteration row by row

1.Initially the distribution is distributed optimally on basis of greedy approach.

2.Using the Knapsack and column generation techniques the duals is calculated (in which the constraints lie behind the idea which sums up to standard length of rod)

3.After that the new cutting patterns are iteratively solved by Linearprogramming Techniques untill a optimal stage is reached.

4.The code snippet suggests which pattern of arbitary length to be used and how much to fullfill the demands.

```python
import numpy as np
from scipy.optimize import linprog

def generate_optimal_pattern(W, w, d):
    w = np.array(w)
    d = np.array(d)

    A = np.eye(len(w)) * (W // w)
    c = np.ones(len(w))
    A_ub = np.atleast_2d(w)

    sol = linprog(c, A_ub=-A, b_ub=-d, bounds=(0, None), meti

    print("A matrix:")
    print(A)
    print("sol.x:")
    print(sol.x)

def solve_knapsack_problems(data):
    for index, (w, d, W) in enumerate(data):
        print(f"Row {index + 1}: w = {w}, d = {d}, W = {W}")
        generate_optimal_pattern(W, [w], [d])
        print("-" * 40)

# Example call to solve_knapsack_problems with data
# data = [(w1, d1, W1), (w2, d2, W2), ...]
# solve_knapsack_problems(data)
```

On combining all the matrices and calculating we can get the demands fullfilled by width as follows-

#For w=1.4
fulfilled=95.89/96
rounded off=96/96

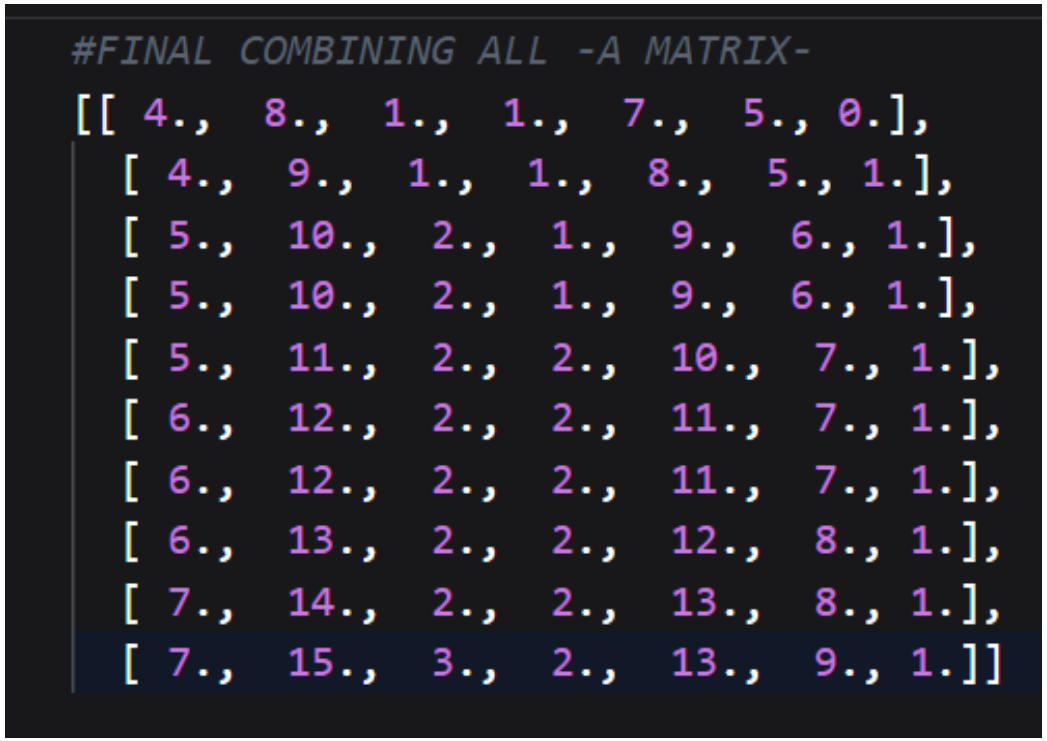
#For w=0.7
fulfilled=20339.9/20340 rounded off=20340/20340

#For w=3.41
fulfilled=1277.99/1278 rounded off=1278/1278
[
#For w=3.98
fulfilled=852/852

#For w=0.765
 rounded off=4970/4970

#For w=1.13
 fulfilled=3975.87/3976 rounded off=3976/3976

#For w=6.18
 fulfilled=2337/2520 [on calculating 90% effective]



ANOTHER APPROACH OF DISTRIBUTING THE DEMANDS BY GENETIC ALGO-Code snippet

1.Snippet uses a genetic algorithm to solve a demand distribution problem.

2.It initializes a population of potential solutions, evaluates their fitness based on a demand (d) and weights (W), and iteratively improves the population through selection,crossover, and mutation.

3.The goal is to find a combination of weights that meets the demand with the minimal total weight.

4.I tried to distribute the demands optimally using Genetic Algorithm.

```python
# Mutation: Bit flip mutation
def mutate(solution, mutation_rate, max_value):
    mutated_solution = solution.copy()
    for i in range(len(mutated_solution)):
        if np.random.rand() < mutation_rate:
            mutated_solution[i] = np.random.randint(0, max_value + 1)
    return mutated_solution
# Genetic Algorithm
def genetic_algorithm(d, W):
    population = initialize_population(POPULATION_SIZE, len(W))
    max_value = int(np.ceil(d / min(W)))

    for generation in range(NUM_GENERATIONS):
        fitness_values = np.array([fitness(individual, d, W) for individual in population])
        new_population = []
        for _ in range(POPULATION_SIZE // 2):
            parents = [tournament_selection(population, fitness_values) for _ in range(2)]
            offspring1, offspring2 = crossover(parents[0], parents[1])
            offspring1 = mutate(offspring1, MUTATION_RATE, max_value)
            offspring2 = mutate(offspring2, MUTATION_RATE, max_value)
            new_population.extend([offspring1, offspring2])
        population = np.array(new_population)
    fitness_values = np.array([fitness(individual, d, W) for individual in population])
    best_solution_idx = np.argmax(fitness_values)
    best_solution = population[best_solution_idx]
    total_weight = np.sum(np.array(best_solution) * np.array(W))

return best_solution, total_weight
```

On computing on basis of the Genetic Algorithm distribution calculating matrix and sol.x-

#For w=1.4
rounded off=96/96

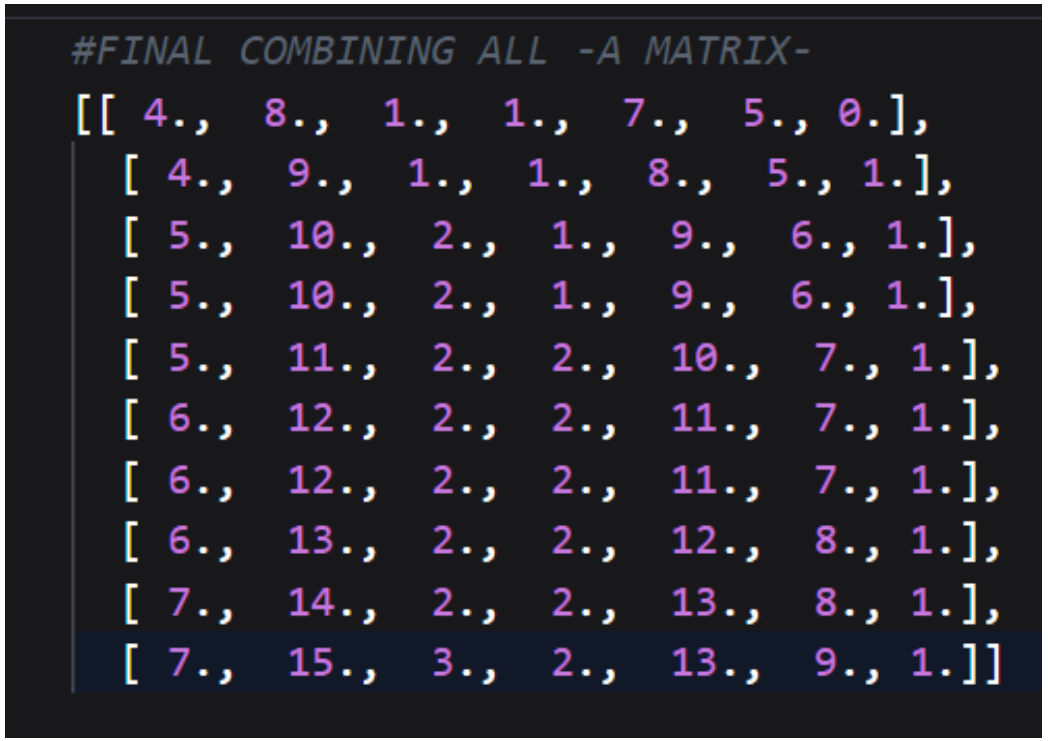
#For w=0.7
fulfilled=19421/20340

#For w=3.41
fulfilled=1278/1278
[
#For w=3.98
fulfilled=827/852

#For w=0.765
rounded off=3326/4970

#For w=1.13
fulfilled=2421/3976

#For w=6.18
fulfilled=2064/2520 [on calculating 86.5% effective]



```
#FINAL COMBINING ALL -A MATRIX-
[[ 4.,  8.,  1.,  1.,  7.,  5., 0.],
  [ 4.,  9.,  1.,  1.,  8.,  5., 1.],
  [ 5.,  10.,  2.,  1.,  9.,  6., 1.],
  [ 5.,  10.,  2.,  1.,  9.,  6., 1.],
  [ 5.,  11.,  2.,  2.,  10.,  7., 1.],
  [ 6.,  12.,  2.,  2.,  11.,  7., 1.],
  [ 6.,  12.,  2.,  2.,  11.,  7., 1.],
  [ 6.,  13.,  2.,  2.,  12.,  8., 1.],
  [ 7.,  14.,  2.,  2.,  13.,  8., 1.],
  [ 7.,  15.,  3.,  2.,  13.,  9., 1.]]
```

ANOTHER APPROACH BY DYNAMIC KNAPSACK TECHNIQUE

#For w=1.4
rounded off=86/96

#For w=0.7
fulfilled=20340/20340

#For w=3.41
fulfilled=1278/1278
[
#For w=3.98
fulfilled=772/852

#For w=0.765
rounded off=4970/4970

#For w=1.13
fulfilled=3976/3976

#For w=6.18
fulfilled=2520/2520 [on calculating 99.7% effective]

```python
import numpy as np

def knapsack_dp(W, w, d):
    n = len(w)
    dp = np.zeros((n + 1, int(W) + 1), dtype=int)

    for i in range(1, n + 1):
        for j in range(1, int(W) + 1):
            if int(w[i - 1]) <= j:
                dp[i][j] = max(dp[i - 1][j], dp[i - 1]
            else:
                dp[i][j] = dp[i - 1][j]

    return dp

def find_selected_items(dp, w, W):
    selected = []
    i = len(w)
    j = int(W)

    while i > 0 and j > 0:
        if dp[i][j] != dp[i - 1][j]:
            selected.append(i - 1)
            j -= int(w[i - 1])
        i -= 1

    return selected
```

Computing the values by Width(W) Inventory data(Code snippet)-

```
    def solve_knapsack(W, w, duals):
        return linprog(-duals, A_ub=np.atleast_2d(w), b_ub=np.atleast_1d(W), bounds=(0, np.inf),
        method='highs')

    for _ in range(1000):
        duals = -sol_dual.ineqlin.marginals
        price_sol = solve_knapsack(total_width, w, duals)
        y = price_sol.x

        if 1 + price_sol.fun < -1e-4:
            A = np.hstack((A, y.reshape((-1, 1))))
            c = np.append(c, 1)
            sol = linprog(c, A_ub=-A, b_ub=-d, bounds=(0, None), method='highs')
        else:
            break

    result = (np.ceil(sol.x).astype(int), A.T)
    return result

# Example usage:
# data = {
#     'w': [1.4, 0.7, 3.41, 3.98, 0.765, 1.13, 6.18],
#     'd': [9, 4, 21, 22, 4, 6, 7]
# }
# df = pd.DataFrame(data)
# total_width = 6.0
# solve_knapsack_problem(df, total_width)
```

COMPARITIVE DIFFERENCES ON THE VARIOUS METHODS INCLUDES-

1.STEPS/APPROACH

2.RESULT

3.REASON BEHIND DIFFERENCE OF RESULTS

ASPECT (STEPS)	GREEDY METHOD	COLUMN GENERATION	KNAPSACK METHOD	GENETIC ALGORITHM
1.INITIALISATION	Sorts rebars based on length.	Starts with initial feasible solution.	Dynamic programming solves subproblem.	Generate initial population of random solutions.
2.SELECTION	Selects best rebar for current need.	Solves restricted master problem.	Define state variables and initialise DP table.	Select individual based on fitness functions.
3.ITERATION	Iterate through sorted rebar and assign cuts.	Solve subproblem to generate new columns.	Choose items based on capacity and constraints.	Perform crossover and mutation generate patterns.
4.EVALUATION	Evaluates local optimally.	Add new column to master and reoptimise.	Calculate optimal solutions for increasing capacity.	Evaluate fitness of each individual in population.
5.REPLACEMENT	No Replacement	No Replacement	No Replacement	Replace old population with new .
6.TERMINATION	Stops when no choices there	Stops when no new columns can improve optimality.	Complete after table is fully populated.	Stops after no. of generation/converge-nce achieved.

Aspect(Result)	Greedy Approach	Knapsack Technique (Dynamic Method)	Genetic Algorithm	Knapsack Technique (Linear Programming)
	Used Demand distribution using Greedy approach.	Calculated the matrix and further calculations	Used Demand Distribution using Genetic Algorithm	Maximise/Mini-mise a linear objective method
	Gave demand fulfillment upto 99.8%	Gave demand fulfillment upto 99.7%	Gave demand fulfillment upto 86.5%	Gave demand fulfillment upto 90%
Reason behind the deviation of results-	Simple fast and easier to implement	Provides optimal solutions for knapsack problems so its better.	Can handle complex problems	Provides optimal solutions for lpp problems.

Challenges-

1.Faced problem in applying and computing matrix and sol.x row by row iteratively.
2.Matching the demands inventory.
3.Computing the final cutting patterns using Genetic Algorithm.

Conclusion-

The 1D Cutting Stock Problem is an important aspect used in Manufacturing.This problem involves various methods of using the mathematical optimisation and other methods to ultimately fullfill demands to minimise the wastage produced which in turn helps in good production,reduce cost and better outcome in industry.

References-

https://towardsdatascience.com/column-generation-in-linear-programming-and-the-cutting-stock-problem-3c697caf4e2b