A HIGH LEVEL DESIGN ON

BundleAI:AnApplicationofMultipleConstraintKnapsack Problem(MCKP)ThroughGeneticAlgorithm(GA)

Submitted in the partial fulfillment of requirements to

CS- 454 – Project Lab

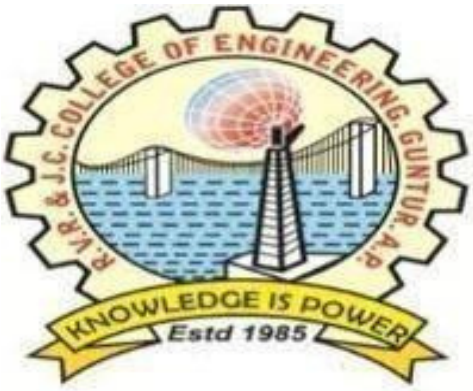
By

Batch No. 14

M.AJAYGOPINATH (Y22CS123)

K.HANUMANTHARAO(Y22CS077)

K.SIVANI BHAVYASRI(Y22CS088)



R.V.R. & J.C. COLLEGE OFENGINEERING (Autonomous)

(NAAC'A+' Grade)

Approved by AICTE :: Affiliatedto Acharya Nagarjuna University

Chandramoulipuram :: Chowdavaram GUNTUR – 522 019, Andhra Pradesh, India.

Mr. Nagulmeera sayyed	**Dr. Z.Sunitha Bai**	**Dr. M. Sreelatha**
Project Guide	Project In-Charge	Prof. & Head,CSE

CONTENTS **PageNo.**

1. Problem Statement

Manual product bundling is a popular marketing strategy that aims to increase the sales of low-selling products by combining them with fast-moving ones. However, most bundling approaches depend on large-scale transactional data, which small businesses lack. Existing optimization strategies often fail to provide scalable, cost-effective, and dynamic bundling solutions without substantial prior data.

Bundle AI addresses this issue by defining product bundling as a Multiple Constraint Knapsack Problem (MCKP) and solving it using a Genetic Algorithm (GA). This system creates optimized product bundles based solely on product information (cost, price, weight, and complementary value) without historical sales data, making it ideal for small-scale enterprises.

Key goals include:

- Automated, data-independent product bundling
- Constraint-aware optimization (weight, cost limits)
- Incorporation of complementary value and reward system
- Efficient and scalable design

2. Functional Requirements

Functionality	Description
Product Data Input	Accepts product attributes like wholesale price, selling price, weight, and complementary score.
Initial Population Setup	Generates initial random product bundles as chromosomes.
Fitness Evaluation	Calculates fitness score using weight, cost constraints, and value-based reward system.
Parent Selection	Selects fitter bundles using roulette-wheel or tournament selection.
Crossover Operation	Combines parts of two parent bundles to generate new offspring bundles.
Mutation Operation	Randomly swaps items within bundles to introduce diversity.
Constrain Handling	Applies penalties to bundles violating weight or cost limits.
Reward Application	Applies rewards for bundles meeting or exceeding complementary thresholds.
Next Generation Evolution	Evolves population iteratively until optimal or near-optimal solution is found.
Output Bundle	Presents best bundle (optimized by weight, cost, and value) to the user.

Product Data Input :

- Collects product attributes like price, weight, and complementary score from the user.

Initial Population Setup

- Creates random combinations of products to form the initial bundle population.

Fitness Evaluation

- Calculates the quality of each bundle using a fitness function.
- Considers constraints like weight, cost, and complementary value during evaluation.

Parent Selection

- Chooses the best-performing bundles based on fitness scores.
- Selected bundles act as parents for producing the next generation.

Crossover Operation

- Combines two parent bundles to create new offspring bundles.
- Enables inheritance of beneficial traits from both parents.

Mutation Operation

- Randomly alters parts of bundles to introduce diversity.
- Helps avoid premature convergence and explore new solutions.

Constraint Handling

- Checks if bundles violate weight or cost constraints.
- Applies penalties to invalid solutions to guide optimization.

Reward Application

- Adds reward scores to bundles that meet or exceed complementary value targets.
- Encourages generation of valuable, customer-friendly bundles.

Output Bundle

- Identifies and extracts the best-performing bundle after all generations.

3. Basic Architecture

3.1.Major Components

Component	Description
Data Input Module	Collects necessary product information for bundling.
Fitness Evaluator	Computes bundle fitness based on price, weight, value, and reward.
Genetic Algorithm (GA) Engine	Core evolutionary solver that initializes, evolves, and optimizes product bundles.
Fitness Evaluator	Computes bundle fitness based on price, weight, value, and reward.
Constraint Manager	Enforces penalties for violating weight and cost limits
Reward System Module	Applies rewards to valuable, complementary bundles.
Output Generator	Displays the best optimized bundle.

A. User Interface (UI)

- **Purpose**: Allows users to input product data and view optimized bundles.
- **Functions**: Collects product details (price, weight, complementary score) through forms and displays output results.

B. Backend (Flask App)

- **Purpose:** Manages communication between the UI and the core algorithm.
- **Functions:** Receives inputs, runs the optimization algorithm, returns results, and serves as a REST API if required.

C. Genetic Algorithm (GA) Engine

- **Purpose:** Drives the evolutionary process to find the best product bundles.
- **Functions:** Initializes population, performs crossover and mutation, applies selection, and evolves bundles over generations.

D. Local Search Module

- **Purpose:** Refines generated solutions using domain-specific local improvements.
- **Functions:** Applies focused optimization within columns and sub-blocks to reduce constraint violations and improve fitness.

E. Fitness Evaluator

- **Purpose:** Assesses how good each bundle is.
- **Functions:** Calculates bundle fitness based on weight, price, complementary score, and applies penalties and rewards.

F. Constraint Manager

- **Purpose:** Ensures solution validity.
- **Functions:** Checks bundles against weight and cost limits, applying penalties for violations to discourage invalid bundles.

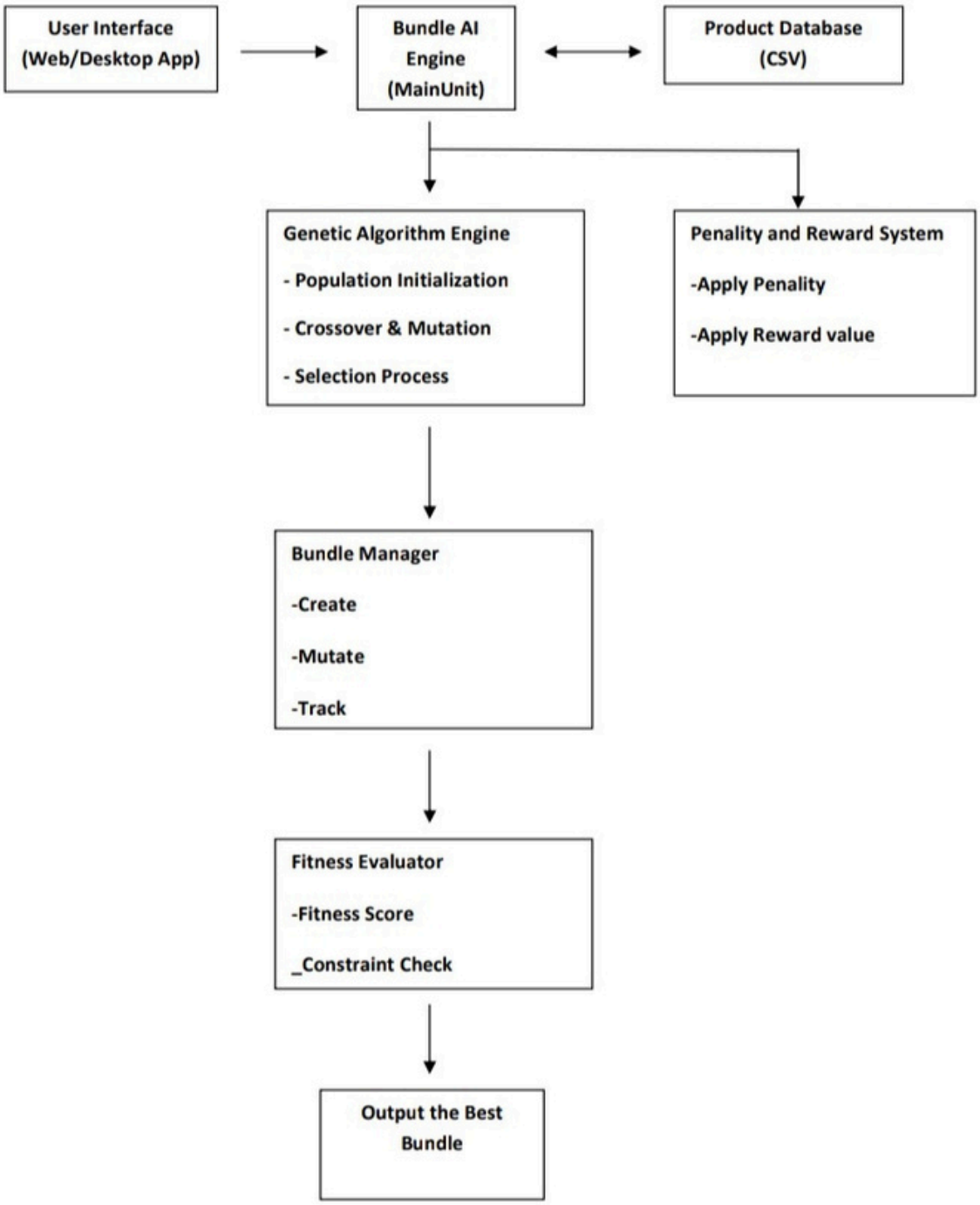
G. Reward System Module

- **Purpose:** Encourages valuable solutions.
- **Functions:** Applies reward scores to bundles that achieve higher complementary values or other target metrics.

H. Output Generator

- **Purpose:** Presents the final result to the user.
- **Functions:** Extracts the best-performing bundle from the algorithm and formats it for display in the UI.

3.2. System Architecture Diagram



3.3. Workflow

Bundle AI starts with collecting product information, including price, weight, and complementary value. This data is used to randomly generate an initial population of product bundles. Each bundle is then evaluated based on fitness, considering constraints and rewards. The best bundles are selected as parents for crossover, producing offspring bundles. Mutation introduces variations for exploration. Bundles are checked for constraints, and rewards are applied where applicable. This evolutionary process continues until an optimal bundle is produced and presented as output.

Below is a step-by-step breakdown of the system's end-to-end workflow:

Step 1: Product information input

The user provides a list of products where each product includes :

- Wholesale Price
- Lowest Selling Price
- Weight (in kilograms)
- Complementary Value Score

Each product is assigned a value type:

- High-selling (h)
- Selling (s)
- Low-selling (l)

These types determine complementary value when grouped:

Complementarity Combination	**Value**
h + l	30
s + l	15
h/s/l (no complementarity)	5

Step 2: Population Initialization

- The algorithm begins by randomly generating initial product bundles, each treated as a chromosome in the genetic algorithm.
- Each bundle is a subset of available products.
- Initial Population Size: 50 (configurable)
- Chromosome Length (li): Randomly between 3 to 5 products per bundle.
- Only combinations within preliminary weight and cost limits are admitted.

Step 3: Fitness Evaluation with Constraints & Rewards

Each bundle is scored using a fitness function that incorporates:

- Selling Price (x): Total of lowest selling prices of items in the bundle
- Cost Constraint (y): Predefined limit, e.g., ₹750
- Weight Constraint: e.g., 1.5 kg
- Penalty (p): Applied if weight or cost exceeds limits
- Reward (r): Based on complementary value thresholds
- Fitness Formula:

  Fitness Score = x/y -p +r

Step 4: Parent Selection

- Parents are selected using Roulette Wheel Selection based on fitness scores.
- Each bundle gets a selection chance proportional to its fitness.
- A random point on the wheel is selected to choose a parent.
- Fitter bundles have a higher chance, but others are not excluded.
- This helps maintain both quality and diversity in the population.
- **Ex: If a bundle has fitness score of 0.3 then it has 30% chance of selection.**

Step 5: Crossover Operation

- Two parent bundles are selected.
- A random crossover point is chosen.
- The offspring bundle inherits products from both parents up to and after the crossover point.

Example:

Parent A: {p1, p3, p6, p7}

Parent B: {p2, p4, p5, p8}

Offspring (after crossover): {p1, p3, p5, p8}

Step 6: Mutation Operation

- Introduces randomness to prevent local optima.
- Mutation Probability: 0.18 (2 in 11 chance).

Mutation Mechanism:

- A random product in the bundle is swapped or replaced with another product from the stock list.
- Only non-duplicate, valid items are used to maintain feasibility.

Step 7: Constraint Check & Reward Application

Weight and cost limits are checked again:

- If Total Weight > 1.5kg → Apply penalty
- If Wholesale Cost > ₹750 → Apply penalty

If within thresholds:

- Reward points (r) are added based on complementarity score.
- Only valid bundles are carried forward to the next generation

Step 8: Evolution to Next Generation

A new generation is formed using:

- Selected Offspring
- Elite Individuals from previous generation
- This process gradually improves average population fitness over generations.

Step 9: Convergence Check

The GA stops when:

- A bundle reaches a fitness score ≥ 1.0, or
- No significant improvement over N generations (e.g., 50 iterations)
- The best-performing bundle is returned.

Step 10: Output Best Bundle

- The final, optimized bundle with maximum fitness score is presented as the output to the user.

3.4. Patterns

Bundle AI uses several patterns to optimize its process. The Constraint-Aware Evolutionary Pattern ensures bundles meet cost and weight limits. The Hybrid Reward-Penalty Pattern rewards valuable bundles and penalizes invalid ones. With the Modular Component Pattern, the system remains organized and easy to update. The Iterative Fitness Loop Pattern refines solutions step-by-step, while the Elitism and Replacement Pattern preserves the best bundles in every generation for consistent improvement.

1. Constraint-Aware Evolutionary Pattern
    - This pattern ensures that the genetic algorithm respects problem-specific constraints, like weight and cost limits, during population evolution. It prevents invalid solutions from dominating the population.

2. Hybrid Reward-Penalty Pattern
    - In this pattern, solutions violating constraints are penalized, while those meeting or exceeding complementary value thresholds are rewarded. This balances exploration and optimization within the algorithm.

3. Modular Component Pattern
    - The system is designed with separate, reusable modules (e.g., input handler, fitness evaluator, reward system), allowing easy maintenance, scalability, and updates to individual components without affecting the entire system

4. Iterative Fitness Loop Pattern
    - This pattern involves continuous evaluation of each generation's bundles using a fitness function in a loop. It helps refine the population iteratively until optimal or near-optimal bundles are generated.

5. Elitism and Replacement Pattern
    - To preserve the best solutions, the algorithm uses elitism, where top-performing bundles are directly carried to the next generation. Remaining slots are filled with new offspring to ensure steady progress toward optimization.

4. Non-Functional Requirements

Non-functional requirements focus on the performance, usability, scalability, reliability, and security aspects of Bundle AI. The system is designed to generate optimized bundles quickly, even for large inventories, ensuring good performance. Usability is ensured through a simple interface requiring only basic product inputs. Its modular design supports easy maintenance and scalability, allowing future updates without major rework. Reliability is maintained through strict constraint handling to ensure valid bundles are always returned. Portability is achieved through Python implementation, making it adaptable across platforms. Finally, input validation ensures correct product data entry, preventing system crashes and ensuring secure operations.

Category	Description
Cost-Efficiency	Works without expensive infrastructure or large-scale data.
Scalability	Supports varying population sizes, cost/weight constraints.
Portability	Pure Python implementation; easy to run on any platform.
Transparency & Explainability	Final output shows product details, value, cost, and fitness score.
Reliability	Validates bundles against cost and weight limits penalizes
Adaptability	Configurable reward thresholds, penalties, and constraints.

5. Technology Stack

Frontend:

- The frontend is optionally built using HTML, CSS, and JavaScript. This allows the user to interact with the system through a simple web interface, where product details can be entered and the optimized bundle results can be displayed clearly. Although Bundle AI can run fully from the console, a frontend enhances usability for non-technical users.

Backend / Logic:

- The core backend logic is implemented in Python, which handles the complete Genetic Algorithm (GA) flow. This includes population initialization, fitness evaluation, parent selection, crossover, mutation, penalty-reward computation, and convergence. Python was chosen for its simplicity, rich library support, and effectiveness in rapid algorithmic development.

Data Storage:

- Product data is read from and written to CSV files. This lightweight format makes it easy to store product attributes such as name, price, cost, weight, and value. It also enables smooth integration with spreadsheets for further analysis or manual editing.

Visualization:

- Matplotlib is used for creating visual outputs such as fitness vs. generation plots. These graphs help users understand how the algorithm converges and how different parameter configurations (like population size) affect performance. This visual feedback is useful for evaluation and tuning.

6. Summary

Bundle AI is an intelligent product bundling system aimed at solving the limitations faced by small-scale businesses that lack access to historical transactional data. The key goals of this project **automated, data-independent product bundling, constraint-aware optimization, complementary value-based bundling, and efficient scalability are effectively achieved through the use of a Genetic Algorithm (GA) model.**

By modeling the bundling process as a Multiple Constraint Knapsack Problem (MCKP), the system ensures that each bundle adheres to specified constraints like maximum cost and weight, making the solution realistic and deployable in real business settings. The use of GA allows the system to automatically generate diverse bundle combinations and evolve them across generations toward optimal solutions.

To enhance bundle value, a reward mechanism is integrated based on the complementary relationship between products encouraging combinations of high-selling and low-selling items. This scoring system increases the practical appeal of the output bundles.

In conclusion, these project objectives are successfully realized through the proposed GA-based model, resulting in a bundling tool that is accurate, scalable, adaptable, and suitable for use in resource-constrained environments.