Constraint-Based Combinatorial Optimization for Smart Outfit Planning with Dynamic Wardrobe Management

Natalia Desiany Nursimin - 13523157
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: nataliadesianyy@gmail.com, 13523157@std.stei.itb.ac.id

***Abstract*—Choosing daily outfits has become an increasingly time-consuming and challenging decision-making task for many individuals, especially when considering factors such as weather conditions, occasion appropriateness, and availability of clean clothing items. Despite owning wardrobes filled with garments, people often struggle with deciding what to wear each day and overlook the numerous outfit combinations that can be created with their existing clothes. To address this challenge, this paper presents an intelligent outfit planning system based on constraint-based combinatorial optimization integrated with dynamic wardrobe management to assist users in efficiently selecting appropriate clothing combinations for daily wear. The system models outfit selection as a combinatorial problem grounded in combinatorics theory, where clothing items are represented as variables with attributes such as category, color, and formality. Constraints include event type, weather conditions, and user-specific preferences, allowing the system to eliminate unsuitable combinations and generate all viable outfit combinations through the usage of combinatoric techniques and efficient constraint satisfaction algorithms. The wardrobe database is dynamically updated to reflect real-time changes, including laundry cycles, newly added garments, and discarded items, ensuring accurate and relevant recommendations. By utilizing the principles of combinatorics and constraint optimization, the system aims to deliver highly personalized, context-aware outfit suggestions that promote style diversity and maximize wardrobe utility. This paper aims to demonstrate how combinatorial techniques combined with constraint-based optimization can be effectively applied to solve real-world problems, particularly in automating personalized decision-making processes such as daily outfit planning.**

***Keywords*—combinatorics; outfit planning; decision making; wardrobe optimization; constraint satisfaction**

I. Introduction

In everyday life, people are constantly faced with making countless decisions, both major and minor. Among these, one surprisingly common and often frustrating dilemma people face is the simple yet persistent question: “What should I wear today?” Although it may seem trivial at first glance, choosing an outfit is far from a simple task, especially when considering multiple factors such as current weather conditions (whether it is sunny, rainy, or windy), the nature of the day’s activities or events, and one’s personal comfort and style preferences. The seemingly mundane task of selecting clothing can take up a significant amount of time and mental energy, particularly when individuals find themselves staring at a wardrobe full of options, yet feeling as though they have nothing suitable to wear.

This paradox of owning many clothes but still struggling to find something to wear is a widespread problem experienced by many people on a daily basis. It is a classic example of decision fatigue, where the mental burden of having to choose from too many options results in people being overwhelmed, frustrated and indecisive. In many cases, people default to a few familiar outfits out of habit or convenience, leaving the majority of their wardrobe untouched. This leads to dissatisfaction with their personal style and, often, to the frequent purchase of new clothes in an attempt to “fix” the problem. Unfortunately, in most cases this often just results in the same cycle repeating itself: the new additions go underutilized, and the daily dilemma and underutilization persists.

The effects that emerge from this cycle not only contribute significantly to individual stress but also to broader issues such as excessive consumption and fashion waste. When individuals repeatedly purchase new clothes without effectively utilizing what they already own, the sustainability of their consumption habits comes into question. This contributes to the growing environmental impact of the fashion industry, which is already a significant source of global waste, increased pollution and resource depletion. Thus, making the issue of solving the outfit planning dilemma not just a matter of convenience, but also a very important matter that ties into responsible consumption, ecological awareness and environmental sustainability.

Moreover, the sense of frustration and dissatisfaction that stems from daily outfit stress can also negatively impact self-image and confidence. Clothing is not as simple as just wearing clothes but also a form of self-expression, and when individuals feel limited or uninspired by what they wear, it may influence their mood and overall sense of identity. Therefore, a system that can intelligently assist in generating outfit combination options is very important, as it not only reduces decision fatigue but can also enhance daily wellbeing and self-esteem.

While existing fashion recommendation systems focus primarily on style matching and trend analysis, few approaches have systematically addressed the combinatorial nature of outfit planning or incorporated dynamic wardrobe state management. Current solutions in the market typically rely on predefined style rules, color theory, or collaborative filtering based on user preferences and fashion trends. However, these systems often fail to consider the practical constraints that real users face, such as weather conditions, occasion appropriateness, and most importantly, the actual availability of clothing items in their personal wardrobes. Furthermore, existing approaches do not account for the temporal dynamics of wardrobe management, such as laundry cycles, considering whether clothing items are clean or dirty. This presents an opportunity to develop a more comprehensive approach to automated outfit planning that considers both the mathematical structure of the problem and constraints that users encounter daily.

It is within this context that the application of mathematical principles, particularly from combinatorics, becomes highly relevant. Combinatorics offers a way to systematically explore all possible combinations of clothing items, helping to uncover all possible outfit pairings that may have never even been considered. With even just a small number of items, there are lots of possible combinations someone could style something. However, this process would become very inefficient and slow if done manually, especially if there are a large number of clothing options. Thus, this is where computational assistance comes in handy.

From a combinatorics perspective, given n tops, m bottoms, and k shoes, there exist $n \times m \times k$ possible outfit combinations. However, constraints significantly reduce this number. For instance, implementing weather and occasion constraints might significantly reduce a lot of these possible combinations.

By designing a program that incorporates constraint-based combinatorial optimization, creating an automated system can be done very quickly. The program can filter out impractical or inappropriate combinations while highlighting those that align with the user's preferences and context. For example, the system can rule out wearing tank tops on a rainy day or suggest more formal combinations for work-related events. By adjusting to each individual's needs, the system's recommendations could prove to be very useful.

In the long term, it could also transform how people interact with their wardrobes. It encourages conscious usage by helping individuals make the most of what they already own, and reduces the perceived need for constant shopping. Ultimately, the use of combinatorial techniques and constraint-based optimization possesses a huge potential to offer a powerful solution to help make people's daily lives easier and encourage sustainable living.

II. THEORETICAL FRAMEWORK

A. Fundamentals of Combinatorics in Outfit Planning

Combinatorics is a fundamental branch of mathematics that deals with counting, arrangement, and selection of objects, often without the need to explicitly list all possible configurations. In the context of smart outfit planning, combinatorial theory provides the mathematical basis for systematically exploring and quantifying the possible combinations of clothing items, filtering them based on specified conditions, and selecting optimal results using constraint-based approaches.

In the context of implementing a smart wardrobe planning system, we can implement the use of combinatorics by treating each clothing item as an element in a finite set. Each outfit can be modeled as a selection of one item from each category, such as top, bottom, outerwear, shoes, and accessories. The total number of outfits then corresponds to the number of possible combinations formed from these subsets.

For example, a basic wardrobe structure can be mathematically represented as follows:

$W = \{T, B, O, S, A\}$ is a wardrobe where:

- $T = \{t_1, t_2, ..., t_n\}$ is the set of tops
- $B = \{b_1, b_2, ..., b_m\}$ is the set of bottoms
- $O = \{o_1, o_2, ..., o_k\}$ is the set of outerwear
- $S = \{s_1, s_2, ..., s_p\}$ is the set of shoes
- $A = \{a_1, a_2, ..., a_r\}$ is the set of accessories

We can calculate the number of possible outfit combinations excluding the use of constraints using combinatorics. For example, a user owns 5 tops, 4 bottoms, 3 outerwears, 2 shoes, and 3 accessories, the number of possible outfit combinations excluding the use of constraints is:

$$\begin{aligned}\text{Total Combinations} &= T \times B \times O \times S \times A \\ &= 5 \times 4 \times 3 \times 2 \times 3 \\ &= 360\end{aligned}$$

B. Rule of Product and Rule of Sum

The basic principles of combinatorics lie in two primary counting rules, which consists of the rule of product and the rule of sum. The rule of product is used when there are multiple independent choices to make. For instance, selecting a top (5 choices), a bottom (4 choices), and a pair of shoes (2 choices) results in $5 \times 4 \times 2 = 40$ possible combinations. Each step is independent, and every option in one category can be paired with every option in the next.

The rule of sum applies when making one choice among multiple categories. For example, if an outfit includes either a jacket (3 options) or a hoodie (2 options), and not both, then there are $3 + 2 = 5$ options that are available for that part of the outfit.

C. *Permutations*

Permutations are used when the order of selection matters. Permutation can be defined as an arrangement of elements that requires a specific order and position. Permutation can be expressed as the following:

$$P(n,r) = \frac{n!}{(n-r)!}$$

Where:

- n = total number of items
- r = number of items selected in order
- ! denotes factorial (for example 4! = 4 x 3 x 2 x 1 = 24)
- $r \leq n$,

D. *Combinations*

Combinations are selections of items from a larger set, where the order of selection does not matter. In contrast to permutations, combinations are concerned only with the elements chosen, not the sequence in which they appear. Combination can be expressed as the following:

$$C(n,r) = \frac{n!}{r!(n-r)!}$$

Where:

- n = the total number of elements in the set
- r = the number of elements selected
- n! = the factorial of n

E. *Repetition and Multisets*

In standard combinations, each item can only be selected once. However, in real-world outfit planning scenarios, certain clothing items can be selected multiple times within a single outfit, such as accessories. Combinations with repetition occur when we are allowed to choose the same item multiple times from a given set. Unlike regular combinations where each element can only be picked once, repetition allows for multiple selections of the same type of item. The number of ways to choose r items from n distinct types with repetition allowed is given by the formula:

$$C(n+r-1,\, r) = \frac{(n+r-1)!}{r!(n-1)!}$$

Where:

- n = number of distinct item types
- r = number of items selected

Multisets are collections of objects where repetition of elements is allowed. In the context of outfit planning, a multiset represents a selection of clothing items where the same type of item can appear multiple times. The number of unique arrangements for a multiset of n items, with repetitions of n1,n2,..,nk. The number of unique arrangements for a multiset of n items can be calculated using the formula below:

$$\frac{n!}{n1!\ x\ n2!\ x\ ....\ x\ nk!}$$

F. *Constraint-Based Combinatorics*

Constraint-based combinatorics is the integration of logical and contextual constraints into the combinatorial process. In this approach, constraints are applied during the generation phase, allowing only valid combinations to be considered from the outset. As a result, the system avoids generating impractical or inappropriate combinations, making the process more efficient and relevant. Constraints are generally divided into two categories, which are hard and soft constraints.

Hard constraints are strict, non-negotiable rules that must be satisfied for a combination to be considered valid. If even one hard constraint is violated, the combination is immediately seen as invalid and will be discarded. In the context of outfit planning, examples of hard constraints include:

- Weather compatibility: No sandals on rainy days
- Occasion requirements: Formal events require formal shirts

Soft constraints are preference-based guidelines. While it is ideal to satisfy them, violating a soft constraint does not make a solution invalid, but may reduce the desirability of the recommendation. In the the context of outfit planning, soft constraints may include:

- Color harmony: Compatibility of outfit colors
- Color preferences: Preferring bright colors over dark ones

G. *Inclusion-Exclusion Principle*

The inclusion-exclusion principle is a fundamental concept in combinatorics used to count the number of elements in the union of multiple sets, especially when these sets overlap. Without this principle, counting overlapping sets may lead to double-counting. This principle can be expressed as the following:

$$|A \cup B| = |A| + |B| - |A \cap B|$$

Where:

- |A| = Number of elements in set A
- |B| = Number of elements in set B
- |A∩B| =Number of elements that are in both A and B

III. Discussion

The daily decision of selecting an outfit, while seemingly trivial, causes a repetitive combinatorial challenge influenced by dynamic and contextual constraints. Despite owning a wide variety of clothing items, individuals often struggle to create appropriate combinations due to factors such as weather conditions, occasion formality, and the real-time availability of clean garments. This section presents an in-depth discussion on the application of constraint-based combinatorial optimization as a rigorous mathematical approach to address this issue. The following subsections explore the practical implementation of this framework within the context of smart wardrobe planning:

1. Combinatorial Representation of the Wardrobe

The foundation of this system lies in representing each clothing category as a finite, disjoint set of items. A wardrobe is defined as the following expression: W = {T, B, O, S, A}, where:

- T = set of tops
- B = set of bottoms
- O = set of outerwear
- S = set of shoes
- A = set of accessories

Outfit generation follows the Rule of Product. If a user has 20 tops, 15 bottoms, and 10 shoes, the number of basic outfit combinations is:

- Basic_Outfits = $|T| \times |B| \times |S| = 20 \times 15 \times 10 = 3{,}000$
- With outerwear included, possible combinations become = $|T| \times |B| \times |O| \times |S| = 20 \times 15 \times 12 \times 10 = 36{,}000$

This illustrates the exponential growth of possibilities and justifies the use of combinatorial theory for optimization.

2. Constraint-Based Filtering of Combinations

Rather than evaluating all combinations and then eliminating invalid ones, the system incorporates constraint satisfaction directly into the generation process. Each outfit combination is filtered using the conjunction of hard constraints:

- Availability: Only includes items marked as available
- Weather Compatibility: Ensures all items are suitable for the specified weather category, for example not using sandals in cold weather.
- Occasion Formality: Filters combinations that meet minimum formality thresholds for different event types

This constraint-based approach can be denote as the following expression:

Valid_Outfits = $|\{(t, b, s) \in T \times B \times S \mid C(t, b, s) = \text{True}\}|$

The function C(t, b, s) represents a constraint function that returns True if the outfit satisfies all required constraints and False if otherwise. This constraint-based approach helps greatly reduces computational time while ensuring practical relevance of recommendations.

3. Advanced Combinatorics for Accessory Integration

Accessories follow a different combinatorial rule set, as multiple accessories can be worn simultaneously. This creates a combination without repetition scenario. For r accessories, the number of ways to select k accessories is:

$$C(r,k) = \frac{r!}{k!(r-k!)}$$

For example, with r = 15 accessories:

- No accessories: C(15,0) = 1 (empty set)
- Single accessory: C(15,1) = 15
- Double accessories: C(15,2) = 105
- Triple accessories: C(15,3) = 455

Total accessory combination: 1+ 15 + 105 + 455 = 576

When combined with 3,000 basic outfit possibilities (top, bottom, shoes):

Total_Complete_Outfits = $3{,}000 \times 576 = 1{,}728{,}000$ possible combinations

To help manage the number of possible combinations, the system applies selective accessory integration, where accessories are only added to the highest-scoring base combinations, balancing computational efficiency with recommendation diversity.

4. Sequential Constraint Application for Efficiency

In order to help enhance computational efficiency of the system, constraints are assigned in the following optimized sequence:

- Stage 1: Availability Filtering → eliminates unavailable items from each category
- Stage 2: Weather Filtering → narrows combinations to seasonally appropriate items
- Stage 3: Occasion Filtering → applies formality logic (e.g., average formality score ≥ 7 for formal events)

Each stage aims to progressively reducs the combinatorial search space to help decrease the amount of possible combinations.

5. Temporal Constraints and Dynamic Availability

The wardrobe's availability is time-dependent, as it reflects real-world dynamics. Each item i has an availability function A(t, i).Time-constrained wardrobe sets is express as:

$T(t) = \{i \in T \mid A(t,i) = \text{True}\}$, B(t), O(t), S(t) follow the same logic

Laundry cycles are modeled as the following:

If a laundry cycle = 7 days and downtime (unavailability) = 2 days, then:

$$P(Available(i)) = (7-2) / 7 = 5 / 7 \approx 71\%$$

This temporal modeling helps enable realistic forecasting of future outfit planning options and accounts for the dynamic nature of wardrobe management in daily life.

IV. IMPLEMENTATION

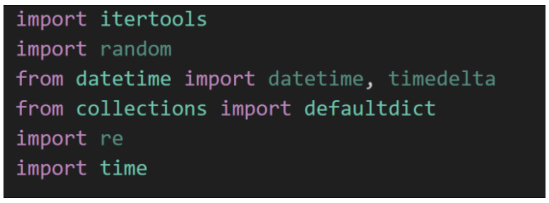
The program has been designed using several important criteria for analyzing outfit appropriate and relevantness, by using contraints such as weather conditions, occasion formality, color harmony, style compatibility, item availability, and personal preferences. Combinatorial theory and constraint satisfaction principles are used to systematically evaluate each condition, enabling outfit recommendations to be generated in the most effective way possible.

A. Import Modules

The program utilizes several essential Python libraries to implement the combinatorial optimization system:

Module Functions:

- itertools: implements combinatorial functions product() and combinations() for systematic outfit generation
- random: provides randomization for outfit sampling and wardrobe simulation
- datetime & timedelta: manages temporal constraints A(t,i) and usage tracking



```python
import itertools
import random
from datetime import datetime, timedelta
from collections import defaultdict
import re
import time
```

Image 4.1. Imported Modules

B. System Architecture

The system architecture of the program is divided into four different components, consisting of the following:

1. Core Data Structures : this component consists of the combinatorial elements representation that forms the mathematical foundation of the wardrobe system.

2. Constraint Satisfaction Engine : this component implements both hard and soft constraint validation to ensure outfit appropriateness and quality ranking.

3. Combinatorial Generator : this component implements the Rule of Product and combinations theory for systematic outfit generation.

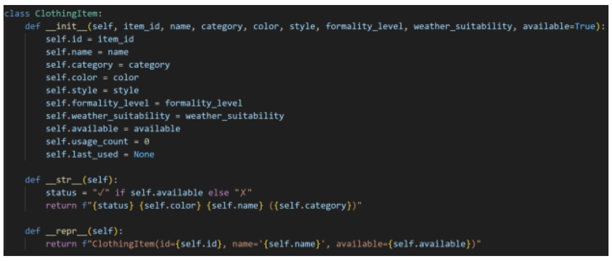
4. User Interface System : this component provides an interactive recommendation interface that translates user preferences into mathematical parameters and presents optimized results in user-friendly formats.

1. Core Data Structures

- ClothingItem Class

  The ClothingItem class represents individual clothing pieces as elements in combinatorial sets, with each item containing attributes that serve as constraints in the optimization process.

  - Each item represents an element in the universal wardrobe set W
  - Availability implements temporal constraint function A(t,i): W → {True, False}
  - Formality level enables numerical constraint evaluation from 1 to 10.



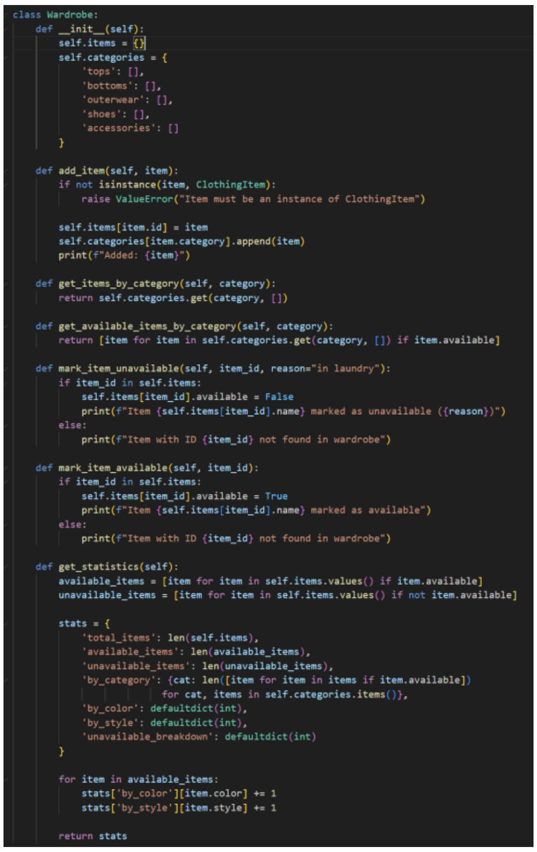
```python
class ClothingItem:
    def __init__(self, item_id, name, category, color, style, formality_level, weather_suitability, available=True):
        self.id = item_id
        self.name = name
        self.category = category
        self.color = color
        self.style = style
        self.formality_level = formality_level
        self.weather_suitability = weather_suitability
        self.available = available
        self.usage_count = 0
        self.last_used = None

    def __str__(self):
        status = "✓" if self.available else "X"
        return f"{status} {self.color} {self.name} ({self.category})"

    def __repr__(self):
        return f"ClothingItem(id={self.id}, name='{self.name}', available={self.available})"
```

Image 4.2. ClothingItem Class

- Wardrobe Class

  The Wardrobe class implements the mathematical structure W = {T, B, O, S, A} as organized disjoint sets with efficient access patterns.

  - add_item(item) : adds a new clothing item to the wardrobe and assigns it to the appropriate category.
  - get_items_by_category(category) : returns all items belonging to a specific clothing category.
  - get_available_items_by_category(category) : returns only available items in a given category
  - mark_item_unavailable(item_id, reason) : marks a specific item as unavailable and logs the reason (for ecample: in laundry)
  - get_statistics() : computes and returns statistics on the total, available, and unavailable items, grouped by category, color, and style.



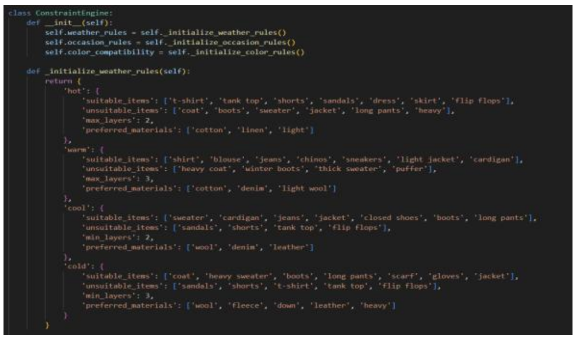
```python
class Wardrobe:
    def __init__(self):
        self.items = {}
        self.categories = {
            'tops': [],
            'bottoms': [],
            'outerwear': [],
            'shoes': [],
            'accessories': []
        }

    def add_item(self, item):
        if not isinstance(item, ClothingItem):
            raise ValueError("Item must be an instance of ClothingItem")

        self.items[item.id] = item
        self.categories[item.category].append(item)
        print(f"Added: {item}")

    def get_items_by_category(self, category):
        return self.categories.get(category, [])

    def get_available_items_by_category(self, category):
        return [item for item in self.categories.get(category, []) if item.available]

    def mark_item_unavailable(self, item_id, reason="in laundry"):
        if item_id in self.items:
            self.items[item_id].available = False
            print(f"Item {self.items[item_id].name} marked as unavailable ({reason})")
        else:
            print(f"Item with ID {item_id} not found in wardrobe")

    def mark_item_available(self, item_id):
        if item_id in self.items:
            self.items[item_id].available = True
            print(f"Item {self.items[item_id].name} marked as available")
        else:
            print(f"Item with ID {item_id} not found in wardrobe")

    def get_statistics(self):
        available_items = [item for item in self.items.values() if item.available]
        unavailable_items = [item for item in self.items.values() if not item.available]

        stats = {
            'total_items': len(self.items),
            'available_items': len(available_items),
            'unavailable_items': len(unavailable_items),
            'by_category': {cat: len([item for item in items if item.available])
                            for cat, items in self.categories.items()},
            'by_color': defaultdict(int),
            'by_style': defaultdict(int),
            'unavailable_breakdown': defaultdict(int)
        }

        for item in available_items:
            stats['by_color'][item.color] += 1
            stats['by_style'][item.style] += 1

        return stats
```

Image 4.3. Wardrobe Class

2. Constraint Satisfaction Engine

This component implements both hard and soft constraint validation to ensure outfit appropriateness and quality ranking. Hard constraints include availability checking A(t,i), weather compatibility validation, occasion formality requirements, and color harmony rules that must be satisfied for valid combinations. Soft constraints provide weighted scoring mechanisms for preference matching and aesthetic quality assessment, enabling multi-objective optimization through scoring functions that rank outfits based on color harmony (70% weight) and personal preferences (30% weight).

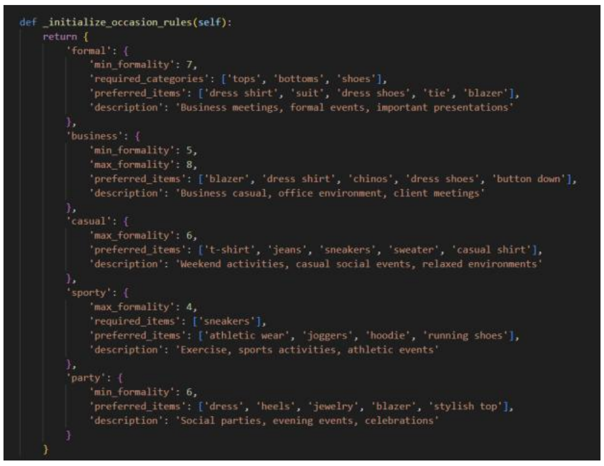
- _initialize_weather_rules(self)
  This function defines weather-based constraints using layer counting and item exclusion rules. The mathematical constraint is implemented as Layer_Count = |{item ∈ outfit : category(item) ∈ {tops, outerwear}}|, where Hot weather requires Layer_Count ≤ 2 and Cold weather requires Layer_Count ≥ 3, ensuring outfit appropriateness for environmental conditions.



```python
class ConstraintEngine:
    def __init__(self):
        self.weather_rules = self._initialize_weather_rules()
        self.occasion_rules = self._initialize_occasion_rules()
        self.color_compatibility = self._initialize_color_rules()

    def _initialize_weather_rules(self):
        return {
            'hot': {
                'suitable_items': ['t-shirt', 'tank top', 'shorts', 'sandals', 'dress', 'skirt', 'flip flops'],
                'unsuitable_items': ['coat', 'boots', 'sweater', 'jacket', 'long pants', 'heavy'],
                'max_layers': 2,
                'preferred_materials': ['cotton', 'linen', 'light']
            },
            'warm': {
                'suitable_items': ['shirt', 'blouse', 'jeans', 'chinos', 'sneakers', 'light jacket', 'cardigan'],
                'unsuitable_items': ['heavy coat', 'winter boots', 'thick sweater', 'puffer'],
                'max_layers': 3,
                'preferred_materials': ['cotton', 'denim', 'light wool']
            },
            'cool': {
                'suitable_items': ['sweater', 'cardigan', 'jeans', 'jacket', 'closed shoes', 'boots', 'long pants'],
                'unsuitable_items': ['sandals', 'shorts', 'tank top', 'flip flops'],
                'min_layers': 2,
                'preferred_materials': ['wool', 'denim', 'leather']
            },
            'cold': {
                'suitable_items': ['coat', 'heavy sweater', 'boots', 'long pants', 'scarf', 'gloves', 'jacket'],
                'unsuitable_items': ['sandals', 'shorts', 't-shirt', 'tank top', 'flip flops'],
                'min_layers': 3,
                'preferred_materials': ['wool', 'fleece', 'down', 'leather', 'heavy']
            }
        }
```

Image 4.4. Weather Constraints

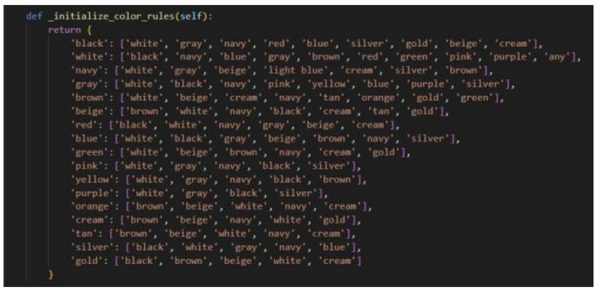
- _initialize_occasion_rules(self)
  This function establishes formality level constraints and item requirements for different occasions using numerical formality scales (1-10) with minimum and maximum thresholds for occasion appropriateness.



```python
def _initialize_occasion_rules(self):
    return {
        'formal': {
            'min_formality': 7,
            'required_categories': ['tops', 'bottoms', 'shoes'],
            'preferred_items': ['dress shirt', 'suit', 'dress shoes', 'tie', 'blazer'],
            'description': 'Business meetings, formal events, important presentations'
        },
        'business': {
            'min_formality': 5,
            'max_formality': 8,
            'preferred_items': ['blazer', 'dress shirt', 'chinos', 'dress shoes', 'button down'],
            'description': 'Business casual, office environment, client meetings'
        },
        'casual': {
            'max_formality': 6,
            'preferred_items': ['t-shirt', 'jeans', 'sneakers', 'sweater', 'casual shirt'],
            'description': 'Weekend activities, casual social events, relaxed environments'
        },
        'sporty': {
            'max_formality': 4,
            'required_items': ['sneakers'],
            'preferred_items': ['athletic wear', 'joggers', 'hoodie', 'running shoes'],
            'description': 'Exercise, sports activities, athletic events'
        },
        'party': {
            'min_formality': 6,
            'preferred_items': ['dress', 'heels', 'jewelry', 'blazer', 'stylish top'],
            'description': 'Social parties, evening events, celebrations'
        }
    }
```

Image 4.5. Occasion Formality Constraints

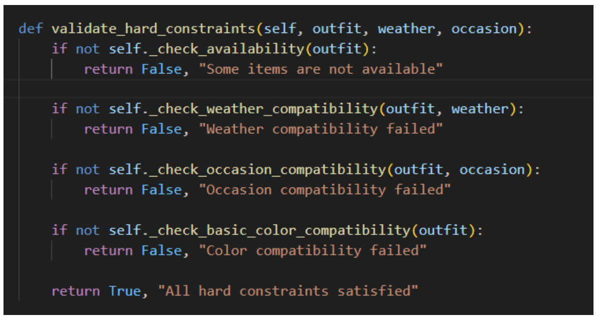
- *_initialize_color_rules(self)*
  This function creates a compatibility matrix defining which colors work harmoniously together, where each color maps to a list of compatible colors with any indicating universal compatibility. The function implements relation C(color1, color2) → {True, False} for pairwise color compatibility checking, serving as the foundation for aesthetic color harmony validation in outfit combinations.



```python
def _initialize_color_rules(self):
    return {
        'black': ['white', 'gray', 'navy', 'red', 'blue', 'silver', 'gold', 'beige', 'cream'],
        'white': ['black', 'navy', 'blue', 'gray', 'brown', 'red', 'green', 'pink', 'purple', 'any'],
        'navy': ['white', 'gray', 'beige', 'light blue', 'cream', 'silver', 'brown'],
        'gray': ['white', 'black', 'navy', 'pink', 'yellow', 'blue', 'purple', 'silver'],
        'brown': ['white', 'beige', 'cream', 'navy', 'tan', 'orange', 'gold', 'green'],
        'beige': ['brown', 'white', 'navy', 'black', 'cream', 'tan', 'gold'],
        'red': ['black', 'white', 'navy', 'gray', 'beige', 'cream'],
        'blue': ['white', 'black', 'gray', 'beige', 'brown', 'navy', 'silver'],
        'green': ['white', 'beige', 'brown', 'navy', 'cream', 'gold'],
        'pink': ['white', 'gray', 'navy', 'black', 'silver'],
        'yellow': ['white', 'gray', 'navy', 'black', 'brown'],
        'purple': ['white', 'gray', 'black', 'silver'],
        'orange': ['brown', 'beige', 'white', 'navy', 'cream'],
        'cream': ['brown', 'beige', 'navy', 'white', 'gold'],
        'tan': ['brown', 'beige', 'white', 'navy', 'cream'],
        'silver': ['black', 'white', 'gray', 'navy', 'blue'],
        'gold': ['black', 'brown', 'beige', 'white', 'cream']
    }
```

Image 4.6. Color Compatibility Matrix

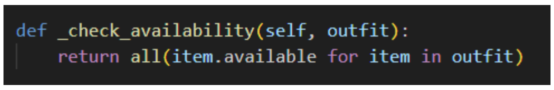
- validate_hard_constraints(outfit, weather, occasion)
  This function orchestrates sequential validation of all hard constraints with early termination, using boolean AND logic (∧) where all constraints must be satisfied for outfit validity. The mathematical logic is implemented as Valid(outfit) = A(outfit) ∧ W(outfit, weather) ∧ O(outfit, occasion) ∧ C(outfit), ensuring comprehensive constraint satisfaction before proceeding to optimization scoring.



```python
def validate_hard_constraints(self, outfit, weather, occasion):
    if not self._check_availability(outfit):
        return False, "Some items are not available"

    if not self._check_weather_compatibility(outfit, weather):
        return False, "Weather compatibility failed"

    if not self._check_occasion_compatibility(outfit, occasion):
        return False, "Occasion compatibility failed"

    if not self._check_basic_color_compatibility(outfit):
        return False, "Color compatibility failed"

    return True, "All hard constraints satisfied"
```

Image 4.7. Hard Constraints Validation

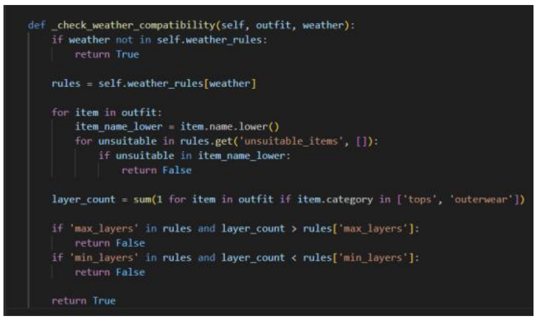
- _check_availability(outfit)
  This function verifies that all items in the outfit are currently available (not in laundry) and ready to be used.



```python
def _check_availability(self, outfit):
    return all(item.available for item in outfit)
```

Image 4.8. Availability Checking Validation

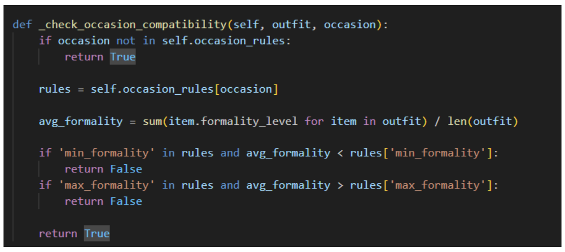
- _check_weather_compatibility(outfit, weather)
  This function ensures outfit appropriateness for specified weather conditions through two-phase validation: unsuitable item detection using string matching and layer count verification against min/max thresholds.



```python
def _check_weather_compatibility(self, outfit, weather):
    if weather not in self.weather_rules:
        return True

    rules = self.weather_rules[weather]

    for item in outfit:
        item_name_lower = item.name.lower()
        for unsuitable in rules.get('unsuitable_items', []):
            if unsuitable in item_name_lower:
                return False

    layer_count = sum(1 for item in outfit if item.category in ['tops', 'outerwear'])

    if 'max_layers' in rules and layer_count > rules['max_layers']:
        return False
    if 'min_layers' in rules and layer_count < rules['min_layers']:
        return False

    return True
```

Image 4.9. Weather Constraint

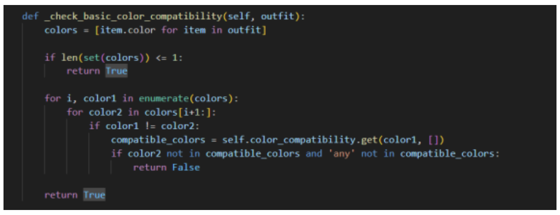
- _ check_occasion_compatibility(outfit, occasion)
  This function validates outfit formality level against occasion requirements using statistical analysis by calculating the arithmetic mean of item formality levels and comparing against occasion-specific thresholds.



```python
def _check_occasion_compatibility(self, outfit, occasion):
    if occasion not in self.occasion_rules:
        return True

    rules = self.occasion_rules[occasion]

    avg_formality = sum(item.formality_level for item in outfit) / len(outfit)

    if 'min_formality' in rules and avg_formality < rules['min_formality']:
        return False
    if 'max_formality' in rules and avg_formality > rules['max_formality']:
        return False

    return True
```

Image 4.10. Occasion Constraint

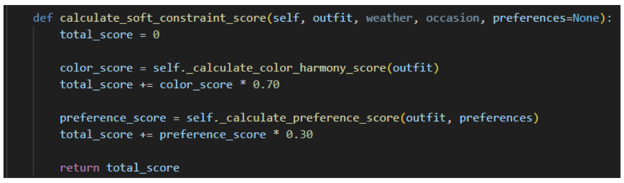
- _ check_basic_color_compatibility(outfit)
  This function ensures color harmony through pairwise compatibility checking using the color matrix, with special case handling for monochromatic outfits (always valid) followed by exhaustive pairwise comparison for multi-color outfits.



```python
def _check_basic_color_compatibility(self, outfit):
    colors = [item.color for item in outfit]

    if len(set(colors)) <= 1:
        return True

    for i, color1 in enumerate(colors):
        for color2 in colors[i+1:]:
            if color1 != color2:
                compatible_colors = self.color_compatibility.get(color1, [])
                if color2 not in compatible_colors and 'any' not in compatible_colors:
                    return False

    return True
```

Image 4.11. Color Compatibility Checking

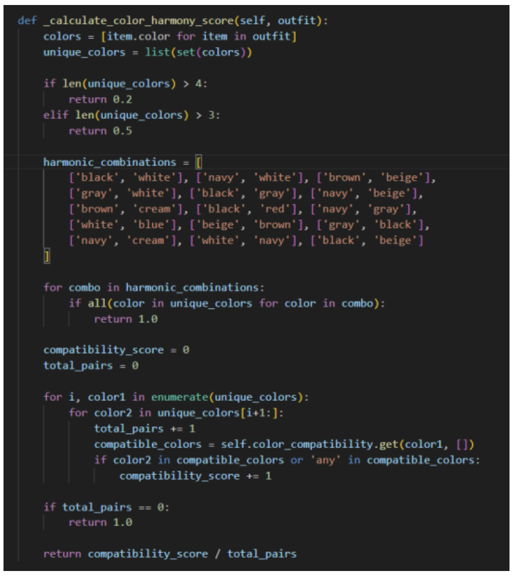
- *calculate_soft_constraint_score(outfit, weather, occasion, preferences)*
  This function implements weighted multi-criteria optimization for outfit quality ranking using a linear combination of aesthetic quality (70%) and personal preference alignment (30%). The mathematical formula $S(outfit) = 0.7 \times S\_color(outfit) + 0.3 \times S\_preference(outfit, preferences)$ balances objective aesthetic principles with subjective user preferences, enabling personalized outfit recommendations while maintaining style coherence.



```python
def calculate_soft_constraint_score(self, outfit, weather, occasion, preferences=None):
    total_score = 0

    color_score = self._calculate_color_harmony_score(outfit)
    total_score += color_score * 0.70

    preference_score = self._calculate_preference_score(outfit, preferences)
    total_score += preference_score * 0.30

    return total_score
```

Image 4.12. Soft Constraints Validation

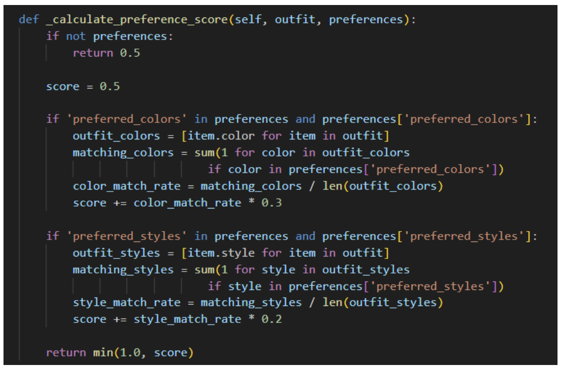
- _calculate_color_harmony_score(self, outfit)



```python
def _calculate_color_harmony_score(self, outfit):
    colors = [item.color for item in outfit]
    unique_colors = list(set(colors))

    if len(unique_colors) > 4:
        return 0.2
    elif len(unique_colors) > 3:
        return 0.5

    harmonic_combinations = [
        ['black', 'white'], ['navy', 'white'], ['brown', 'beige'],
        ['gray', 'white'], ['black', 'gray'], ['navy', 'beige'],
        ['brown', 'cream'], ['black', 'red'], ['navy', 'gray'],
        ['white', 'blue'], ['beige', 'brown'], ['gray', 'black'],
        ['navy', 'cream'], ['white', 'navy'], ['black', 'beige']
    ]

    for combo in harmonic_combinations:
        if all(color in unique_colors for color in combo):
            return 1.0

    compatibility_score = 0
    total_pairs = 0

    for i, color1 in enumerate(unique_colors):
        for color2 in unique_colors[i+1:]:
            total_pairs += 1
            compatible_colors = self.color_compatibility.get(color1, [])
            if color2 in compatible_colors or 'any' in compatible_colors:
                compatibility_score += 1

    if total_pairs == 0:
        return 1.0

    return compatibility_score / total_pairs
```

Image 4.13. Calculate Color Aesthethic Analysis

- _calculate_preference_score(outfit, preferences)
  This function quantifies alignment between outfit characteristics and user preferences by calculating match rates for preferred colors and styles, then applying weighted bonuses to a base score.



```python
def _calculate_preference_score(self, outfit, preferences):
    if not preferences:
        return 0.5

    score = 0.5

    if 'preferred_colors' in preferences and preferences['preferred_colors']:
        outfit_colors = [item.color for item in outfit]
        matching_colors = sum(1 for color in outfit_colors
                              if color in preferences['preferred_colors'])
        color_match_rate = matching_colors / len(outfit_colors)
        score += color_match_rate * 0.3

    if 'preferred_styles' in preferences and preferences['preferred_styles']:
        outfit_styles = [item.style for item in outfit]
        matching_styles = sum(1 for style in outfit_styles
                              if style in preferences['preferred_styles'])
        style_match_rate = matching_styles / len(outfit_styles)
        score += style_match_rate * 0.2

    return min(1.0, score)
```

Image 4.14. Personal Preference Matching

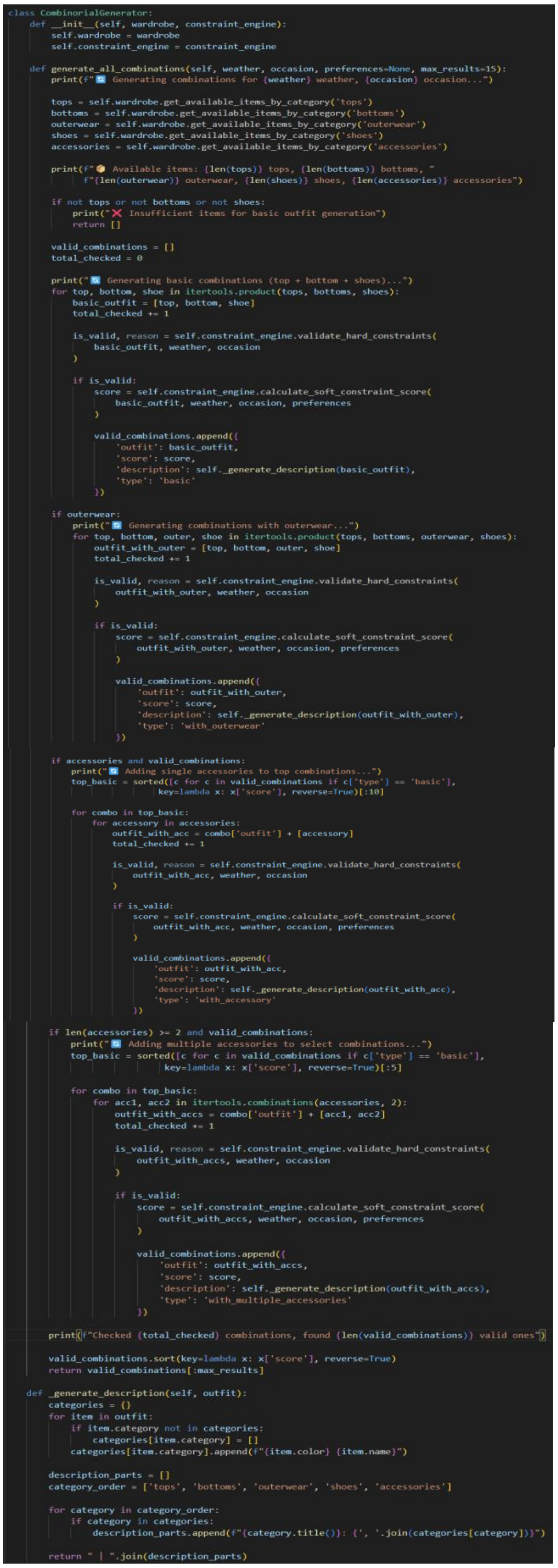
3. Combinatorial Generation Engine

The OutfitGenerator class implements the mathematical combinatorics principles for systematic outfit exploration through three distinct phases. The generator uses the Rule of Product and Combinations theory to create comprehensive outfit possibilities while maintaining computational efficiency through sequential constraint application. This component is divided into three phases, phase 1, phase 2, and phase 3.

Phase 1 consists of the basic combinations which implements the fundamental Rule of Product $|T| \times |B| \times |S|$ where each outfit consists of one top, one bottom, and one shoe. Using itertools.product(), the system generates all possible three-item combinations. Each combination undergoes immediate hard constraint validation for availability, weather compatibility, occasion formality, and color harmony before receiving soft constraint scoring.

Building on the continuation of phase 1, phase 2 consists of all the extended combinations which expands the mathematical model to $|T| \times |B| \times |O| \times |S|$ by incorporating outerwear items, generating a lot more possible combiinations. This phase prioritizes weather-dependent scenarios where additional layers are required, particularly for cool and cold weather conditions. The four-element product demonstrates the exponential growth of combinatorial possibilities while maintaining systematic evaluation.

Lastly, phase 3 which consists of integrating accessories into the system. This phase enriches the top-ranked outfit combinations by applying combinatorial selection using the formula C(n, r) to generate additional styling options. To prevent combinatorial explosion, accessory enhancement is selectively applied only to the top highest-scoring base combinations.



```python
class CombinorialGenerator:
    def __init__(self, wardrobe, constraint_engine):
        self.wardrobe = wardrobe
        self.constraint_engine = constraint_engine

    def generate_all_combinations(self, weather, occasion, preferences=None, max_results=15):
        print(f"🔄 Generating combinations for {weather} weather, {occasion} occasion...")

        tops = self.wardrobe.get_available_items_by_category('tops')
        bottoms = self.wardrobe.get_available_items_by_category('bottoms')
        outerwear = self.wardrobe.get_available_items_by_category('outerwear')
        shoes = self.wardrobe.get_available_items_by_category('shoes')
        accessories = self.wardrobe.get_available_items_by_category('accessories')

        print(f"📦 Available items: {len(tops)} tops, {len(bottoms)} bottoms, "
              f"{len(outerwear)} outerwear, {len(shoes)} shoes, {len(accessories)} accessories")

        if not tops or not bottoms or not shoes:
            print("❌ Insufficient items for basic outfit generation")
            return []

        valid_combinations = []
        total_checked = 0

        print("🔄 Generating basic combinations (top + bottom + shoes)...")
        for top, bottom, shoe in itertools.product(tops, bottoms, shoes):
            basic_outfit = [top, bottom, shoe]
            total_checked += 1

            is_valid, reason = self.constraint_engine.validate_hard_constraints(
                basic_outfit, weather, occasion
            )

            if is_valid:
                score = self.constraint_engine.calculate_soft_constraint_score(
                    basic_outfit, weather, occasion, preferences
                )

                valid_combinations.append({
                    'outfit': basic_outfit,
                    'score': score,
                    'description': self._generate_description(basic_outfit),
                    'type': 'basic'
                })

        if outerwear:
            print("🔄 Generating combinations with outerwear...")
            for top, bottom, outer, shoe in itertools.product(tops, bottoms, outerwear, shoes):
                outfit_with_outer = [top, bottom, outer, shoe]
                total_checked += 1

                is_valid, reason = self.constraint_engine.validate_hard_constraints(
                    outfit_with_outer, weather, occasion
                )

                if is_valid:
                    score = self.constraint_engine.calculate_soft_constraint_score(
                        outfit_with_outer, weather, occasion, preferences
                    )

                    valid_combinations.append({
                        'outfit': outfit_with_outer,
                        'score': score,
                        'description': self._generate_description(outfit_with_outer),
                        'type': 'with_outerwear'
                    })

        if accessories and valid_combinations:
            print("🔄 Adding single accessories to top combinations...")
            top_basic = sorted([c for c in valid_combinations if c['type'] == 'basic'],
                               key=lambda x: x['score'], reverse=True)[:10]

            for combo in top_basic:
                for accessory in accessories:
                    outfit_with_acc = combo['outfit'] + [accessory]
                    total_checked += 1

                    is_valid, reason = self.constraint_engine.validate_hard_constraints(
                        outfit_with_acc, weather, occasion
                    )

                    if is_valid:
                        score = self.constraint_engine.calculate_soft_constraint_score(
                            outfit_with_acc, weather, occasion, preferences
                        )

                        valid_combinations.append({
                            'outfit': outfit_with_acc,
                            'score': score,
                            'description': self._generate_description(outfit_with_acc),
                            'type': 'with_accessory'
                        })

        if len(accessories) >= 2 and valid_combinations:
            print("🔄 Adding multiple accessories to select combinations...")
            top_basic = sorted([c for c in valid_combinations if c['type'] == 'basic'],
                               key=lambda x: x['score'], reverse=True)[:5]

            for combo in top_basic:
                for acc1, acc2 in itertools.combinations(accessories, 2):
                    outfit_with_accs = combo['outfit'] + [acc1, acc2]
                    total_checked += 1

                    is_valid, reason = self.constraint_engine.validate_hard_constraints(
                        outfit_with_accs, weather, occasion
                    )

                    if is_valid:
                        score = self.constraint_engine.calculate_soft_constraint_score(
                            outfit_with_accs, weather, occasion, preferences
                        )

                        valid_combinations.append({
                            'outfit': outfit_with_accs,
                            'score': score,
                            'description': self._generate_description(outfit_with_accs),
                            'type': 'with_multiple_accessories'
                        })

        print(f"Checked {total_checked} combinations, found {len(valid_combinations)} valid ones")

        valid_combinations.sort(key=lambda x: x['score'], reverse=True)
        return valid_combinations[:max_results]

    def _generate_description(self, outfit):
        categories = {}
        for item in outfit:
            if item.category not in categories:
                categories[item.category] = []
            categories[item.category].append(f"{item.color} {item.name}")

        description_parts = []
        category_order = ['tops', 'bottoms', 'outerwear', 'shoes', 'accessories']

        for category in category_order:
            if category in categories:
                description_parts.append(f"{category.title()}: {', '.join(categories[category])}")

        return " | ".join(description_parts)
```

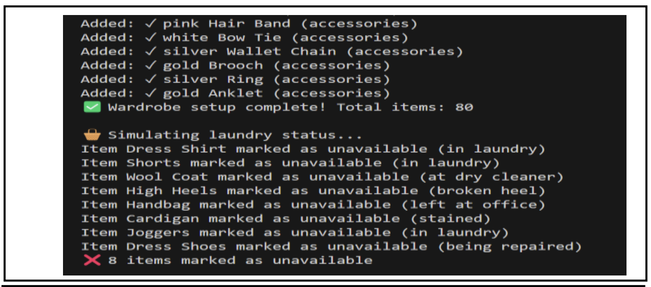
Image 4.15. Combinatorial Generation Engine

4. User Interface System

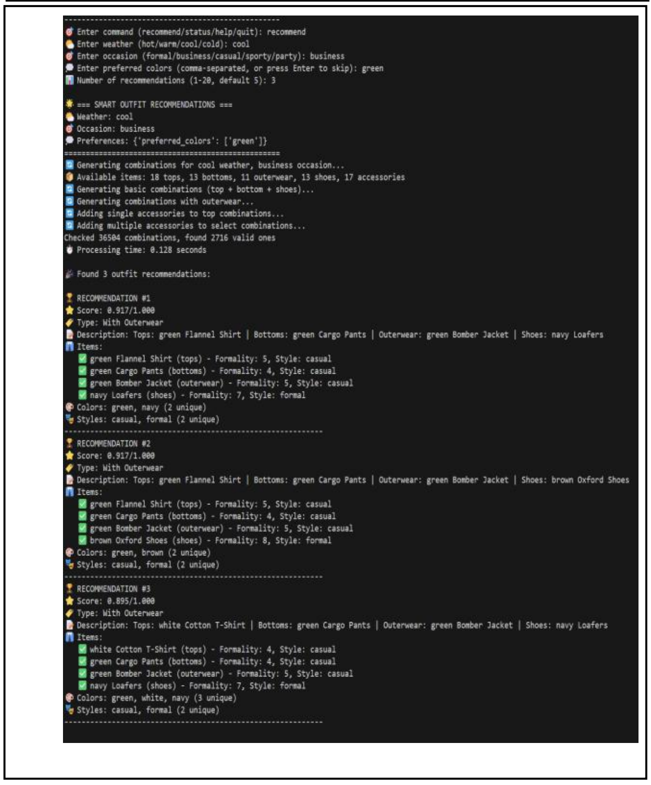
The User Interface System is used as the interface component that connects users with the underlying mathematical engine. This component is responsible for things such as:

- Input Processing: receiving user inputs (weather, occasion, color preferences) and converting them into mathematical parameters for the optimization engine
- Command Management: managing interactive commands (recommend/status/help/quit) with comprehensive input validation
- Output Presentation: displaying recommendation results in user-friendly formats, including outfit scores (0-1.000), color or style analysis, and item details
- System Monitoring: providing wardrobe status, item availability statistics, and system performance information

V. TESTING & RESULTS



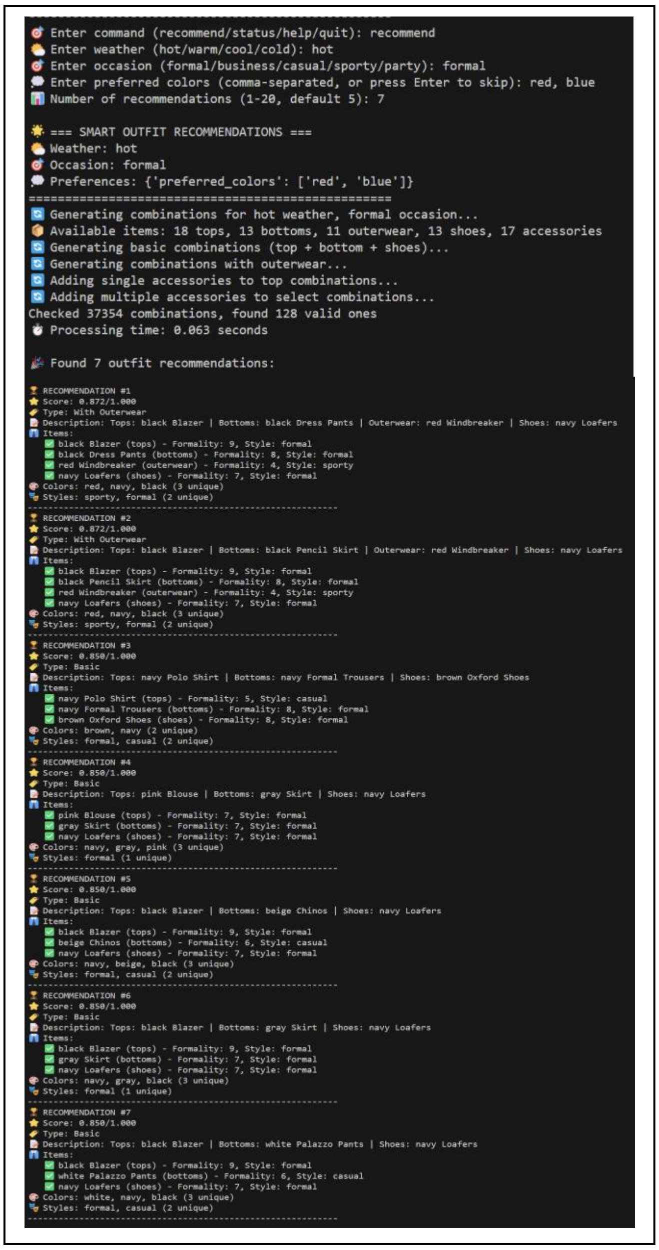
```
Added: ✓ pink Hair Band (accessories)
Added: ✓ white Bow Tie (accessories)
Added: ✓ silver Wallet Chain (accessories)
Added: ✓ gold Brooch (accessories)
Added: ✓ silver Ring (accessories)
Added: ✓ gold Anklet (accessories)
✅ Wardrobe setup complete! Total items: 80

🧺 Simulating laundry status...
Item Dress Shirt marked as unavailable (in laundry)
Item Shorts marked as unavailable (in laundry)
Item Wool Coat marked as unavailable (at dry cleaner)
Item High Heels marked as unavailable (broken heel)
Item Handbag marked as unavailable (left at office)
Item Cardigan marked as unavailable (stained)
Item Joggers marked as unavailable (in laundry)
Item Dress Shoes marked as unavailable (being repaired)
❌ 8 items marked as unavailable
```



```
--------------------------------------------------
🎯 Enter command (recommend/status/help/quit): recommend
🌤 Enter weather (hot/warm/cool/cold): cool
🎯 Enter occasion (formal/business/casual/sporty/party): business
💭 Enter preferred colors (comma-separated, or press Enter to skip): green
📊 Number of recommendations (1-20, default 5): 3

🌟 === SMART OUTFIT RECOMMENDATIONS ===
🌤 Weather: cool
🎯 Occasion: business
💭 Preferences: {'preferred_colors': ['green']}
==================================================
🔄 Generating combinations for cool weather, business occasion...
📦 Available items: 18 tops, 13 bottoms, 11 outerwear, 13 shoes, 17 accessories
🔄 Generating basic combinations (top + bottom + shoes)...
🔄 Generating combinations with outerwear...
🔄 Adding single accessories to top combinations...
🔄 Adding multiple accessories to select combinations...
Checked 36504 combinations, found 2716 valid ones
⏱ Processing time: 0.128 seconds

🎉 Found 3 outfit recommendations:

🏆 RECOMMENDATION #1
⭐ Score: 0.917/1.000
🏷 Type: With Outerwear
📄 Description: Tops: green Flannel Shirt | Bottoms: green Cargo Pants | Outerwear: green Bomber Jacket | Shoes: navy Loafers
👖 Items:
   ✅ green Flannel Shirt (tops) - Formality: 5, Style: casual
   ✅ green Cargo Pants (bottoms) - Formality: 4, Style: casual
   ✅ green Bomber Jacket (outerwear) - Formality: 5, Style: casual
   ✅ navy Loafers (shoes) - Formality: 7, Style: formal
🎨 Colors: green, navy (2 unique)
🎭 Styles: casual, formal (2 unique)
------------------------------------------------------------
🏆 RECOMMENDATION #2
⭐ Score: 0.917/1.000
🏷 Type: With Outerwear
📄 Description: Tops: green Flannel Shirt | Bottoms: green Cargo Pants | Outerwear: green Bomber Jacket | Shoes: brown Oxford Shoes
👖 Items:
   ✅ green Flannel Shirt (tops) - Formality: 5, Style: casual
   ✅ green Cargo Pants (bottoms) - Formality: 4, Style: casual
   ✅ green Bomber Jacket (outerwear) - Formality: 5, Style: casual
   ✅ brown Oxford Shoes (shoes) - Formality: 8, Style: formal
🎨 Colors: green, brown (2 unique)
🎭 Styles: casual, formal (2 unique)
------------------------------------------------------------
🏆 RECOMMENDATION #3
⭐ Score: 0.895/1.000
🏷 Type: With Outerwear
📄 Description: Tops: white Cotton T-Shirt | Bottoms: green Cargo Pants | Outerwear: green Bomber Jacket | Shoes: navy Loafers
👖 Items:
   ✅ white Cotton T-Shirt (tops) - Formality: 4, Style: casual
   ✅ green Cargo Pants (bottoms) - Formality: 4, Style: casual
   ✅ green Bomber Jacket (outerwear) - Formality: 5, Style: casual
   ✅ navy Loafers (shoes) - Formality: 7, Style: formal
🎨 Colors: green, white, navy (3 unique)
🎭 Styles: casual, formal (2 unique)
------------------------------------------------------------
```

Image 5.1. Test Result 1

Test result 1 shows a successful implementation of constraint-based combinatorial optimization. System was shown processing a total of 36,504 possible combinations in 0.128 seconds and identifying 2,716 valid outfits that satisfy all hard constraints. For constraints cool weather, business occasion and green preference scenario, the system correctly applies weather constraints by including outerwear layers, maintains business-appropriate formality levels (4-5.75 average), and prioritizes green color preferences in the optimization scoring. The top recommendations achieve 0.917/1.000 scores through effective color harmony (green-navy, green-brown combinations).



```
🎯 Enter command (recommend/status/help/quit): recommend
🌤 Enter weather (hot/warm/cool/cold): hot
🎯 Enter occasion (formal/business/casual/sporty/party): formal
💭 Enter preferred colors (comma-separated, or press Enter to skip): red, blue
📊 Number of recommendations (1-20, default 5): 7

🌟 === SMART OUTFIT RECOMMENDATIONS ===
🌤 Weather: hot
🎯 Occasion: formal
💭 Preferences: {'preferred_colors': ['red', 'blue']}
==================================================
🔄 Generating combinations for hot weather, formal occasion...
📦 Available items: 18 tops, 13 bottoms, 11 outerwear, 13 shoes, 17 accessories
🔄 Generating basic combinations (top + bottom + shoes)...
🔄 Generating combinations with outerwear...
🔄 Adding single accessories to top combinations...
🔄 Adding multiple accessories to select combinations...
Checked 37354 combinations, found 128 valid ones
⏱ Processing time: 0.063 seconds

🎉 Found 7 outfit recommendations:

🏆 RECOMMENDATION #1
⭐ Score: 0.872/1.000
🏷 Type: With Outerwear
📄 Description: Tops: black Blazer | Bottoms: black Dress Pants | Outerwear: red Windbreaker | Shoes: navy Loafers
👖 Items:
   ✅ black Blazer (tops) - Formality: 9, Style: formal
   ✅ black Dress Pants (bottoms) - Formality: 8, Style: formal
   ✅ red Windbreaker (outerwear) - Formality: 4, Style: sporty
   ✅ navy Loafers (shoes) - Formality: 7, Style: formal
🎨 Colors: red, navy, black (3 unique)
🎭 Styles: sporty, formal (2 unique)
------------------------------------------------------------
🏆 RECOMMENDATION #2
⭐ Score: 0.872/1.000
🏷 Type: With Outerwear
📄 Description: Tops: black Blazer | Bottoms: black Pencil Skirt | Outerwear: red Windbreaker | Shoes: navy Loafers
👖 Items:
   ✅ black Blazer (tops) - Formality: 9, Style: formal
   ✅ black Pencil Skirt (bottoms) - Formality: 8, Style: formal
   ✅ red Windbreaker (outerwear) - Formality: 4, Style: sporty
   ✅ navy Loafers (shoes) - Formality: 7, Style: formal
🎨 Colors: red, navy, black (3 unique)
🎭 Styles: sporty, formal (2 unique)
------------------------------------------------------------
🏆 RECOMMENDATION #3
⭐ Score: 0.850/1.000
🏷 Type: Basic
📄 Description: Tops: navy Polo Shirt | Bottoms: navy Formal Trousers | Shoes: brown Oxford Shoes
👖 Items:
   ✅ navy Polo Shirt (tops) - Formality: 5, Style: casual
   ✅ navy Formal Trousers (bottoms) - Formality: 8, Style: formal
   ✅ brown Oxford Shoes (shoes) - Formality: 8, Style: formal
🎨 Colors: brown, navy (2 unique)
🎭 Styles: formal, casual (2 unique)
------------------------------------------------------------
🏆 RECOMMENDATION #4
⭐ Score: 0.850/1.000
🏷 Type: Basic
📄 Description: Tops: pink Blouse | Bottoms: gray Skirt | Shoes: navy Loafers
👖 Items:
   ✅ pink Blouse (tops) - Formality: 7, Style: formal
   ✅ gray Skirt (bottoms) - Formality: 7, Style: formal
   ✅ navy Loafers (shoes) - Formality: 7, Style: formal
🎨 Colors: navy, gray, pink (3 unique)
🎭 Styles: formal (1 unique)
------------------------------------------------------------
🏆 RECOMMENDATION #5
⭐ Score: 0.850/1.000
🏷 Type: Basic
📄 Description: Tops: black Blazer | Bottoms: beige Chinos | Shoes: navy Loafers
👖 Items:
   ✅ black Blazer (tops) - Formality: 9, Style: formal
   ✅ beige Chinos (bottoms) - Formality: 6, Style: casual
   ✅ navy Loafers (shoes) - Formality: 7, Style: formal
🎨 Colors: navy, beige, black (3 unique)
🎭 Styles: formal, casual (2 unique)
------------------------------------------------------------
🏆 RECOMMENDATION #6
⭐ Score: 0.850/1.000
🏷 Type: Basic
📄 Description: Tops: black Blazer | Bottoms: gray Skirt | Shoes: navy Loafers
👖 Items:
   ✅ black Blazer (tops) - Formality: 9, Style: formal
   ✅ gray Skirt (bottoms) - Formality: 7, Style: formal
   ✅ navy Loafers (shoes) - Formality: 7, Style: formal
🎨 Colors: navy, gray, black (3 unique)
🎭 Styles: formal (1 unique)
------------------------------------------------------------
🏆 RECOMMENDATION #7
⭐ Score: 0.850/1.000
🏷 Type: Basic
📄 Description: Tops: black Blazer | Bottoms: white Palazzo Pants | Shoes: navy Loafers
👖 Items:
   ✅ black Blazer (tops) - Formality: 9, Style: formal
   ✅ white Palazzo Pants (bottoms) - Formality: 6, Style: casual
   ✅ navy Loafers (shoes) - Formality: 7, Style: formal
🎨 Colors: white, navy, black (3 unique)
🎭 Styles: formal, casual (2 unique)
------------------------------------------------------------
```

Image 5.2. Test Result 2

Test Result 2 shows that the smart outfit planner has successfully generated 7 formal outfit recommendations for hot weather, based on user preferences for red and blue colors. Out of 37,354 combinations, 128 valid outfits were found, and the top ones were presented. The highest-ranked outfits (score: 0.872) included black blazers, formal bottoms, red windbreakers, and navy loafers, perfectly aligning with both the formal occasion and user's color preferences. Overall, the system demonstrated accurate filtering and color matching with incredible speed.

VI. CONCLUSION

This paper has successfully demonstrated that combinatorics, when integrated with constraint-based optimization, offers a powerful and practical solution to the everyday problem of outfit planning. By modeling wardrobe selection as a combinatorial problem and applying both hard and soft constraints, the system can effectively narrow down millions of potential combinations into personalized, context-aware recommendations. The results confirm that combinatorial techniques not only improve decision-making efficiency but also support sustainable wardrobe usage by maximizing the value of existing clothing items. Thus, combinatorics proves to be a valuable mathematical tool for solving real-life optimization challenges in a structured, effective and intelligent manner.

VII. APPENDIX

The GitHub repository for this paper can be accessed at https://github.com/nataliadesiany/MakalahMatdis.git . While the video presentation of this paper is available on YouTube and can be accessed at https://youtu.be/A0huBY0gV18?si=EPK0RN-3vD_7LfZ0

VII. ACKNOWLEDGMENT

First and foremost, the author wishes to express heartfelt gratitude to God Almighty for His blessings and grace, which have made it possible to complete this paper on time. The author would also like to express their deepest appreciation to Dr. Ir. Rinaldi Munir, M.T., for his invaluable guidance and all the knowledge he has shared throughout the IF2120 Discrete Mathematics course. In addition, the author extends sincere thanks to their parents for their unwavering support and encouragement during the preparation of this paper. Lastly, the author wishes to thank all readers of this paper and sincerely hopes that the content presented will prove to be both useful and insightful.

REFERENCES

[1] Munir, R. (2024). "Kombinatorika Bagian 1." https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/18-Kombinatorika-Bagian1-2024.pdf.
[Accessed: 17 June 2025].

[2] Munir, R. (2024). "Kombinatorika Bagian 2." https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/19-Kombinatorika-Bagian2-2024.pdf.
[Accessed: 17 June 2025].

[3] Yanuarsih, T., & Arifin, M. (2024). "Implementasi model machine learning 'Style Quest' untuk rekomendasi pakaian berbasis kecerdasan buatan."
https://ejournal.penerbitjurnal.com/index.php/multilingual/article/view/846.
[Accessed: 17 June 2025].

[4] Hayuningtyas, R. Y. (2019). "Penerapan algoritma Naïve Bayes untuk rekomendasi pakaian wanita."
https://doi.org/10.31294/ji.v6i1.4685.
[Accessed: 17 June 2025].

[5] Tang, W., Tang, J., & Tan, C. (2010). "Expertise Matching via Constraint-Based Optimization." In 2010 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (Vol. 1, pp. 34–41). https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=6e48be7fb3484708151603b74f2ca55362610bc6.
[Accessed: 17 June 2025].

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 19 Juni 2025

[illegible]

Natalia Desiany Nursimin
13523157