



Store Recommendation and Route Planning System for Improving Shopping Experience of Users

Bhathiya Vandebona

Index No: 20001924

Harsha Gunawardana

Index No: 20000669

Pubudu Satharasinghe

Index No: 20001665

Supervisor: **Prof. K.P. Hewagamage**

Co-Supervisor: **Mrs. S.S. Thrimahavithana**

April 2025

Submitted in partial fulfillment of the requirements of the
B.Sc in Software Engineering Final Year Project (SCS4223)



Declaration

I certify that this dissertation does not incorporate, without acknowledgement, any material previously submitted for a degree or diploma in any university and to the best of my knowledge and belief, it does not contain any material previously published or written by another person or myself except where due reference is made in the text. I also hereby give consent for my dissertation, if accepted, to be made available for photocopying and for interlibrary loans, and for the title and abstract to be made available to outside organizations.

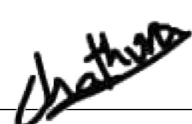
Candidate Name: Bhathiya Vandebona

Signature of Candidate: _____

Date: 30/06/2025

I certify that this dissertation does not incorporate, without acknowledgement, any material previously submitted for a degree or diploma in any university and to the best of my knowledge and belief, it does not contain any material previously published or written by another person or myself except where due reference is made in the text. I also hereby give consent for my dissertation, if accepted, to be made available for photocopying and for interlibrary loans, and for the title and abstract to be made available to outside organizations.


Candidate Name: Harsha Gunawardana

Signature of Candidate: _____

Date: 30/06/2025

I certify that this dissertation does not incorporate, without acknowledgement, any material previously submitted for a degree or diploma in any university and to the best of my knowledge and belief, it does not contain any material previously published or written by another person or myself except where due reference is made in the text. I also hereby give consent for my dissertation, if accepted, to be made available for photocopying and for interlibrary loans, and for the title and abstract to be made available to outside organizations.

Candidate Name: Pubudu Satharasinghe

Signature of Candidate:  _____

Date: 30/06/2025

This is to certify that this dissertation is based on the work of:

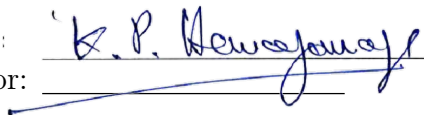
Mr. Bhathiya Vandebona

Mr. Harsha Gunawardana

Mr. Pubudu Satharasinghe


The thesis has been prepared according to the format stipulated and is of acceptable standard.

Principal Supervisor's Name: Prof. K.P. Hewagamage

Signature of Supervisor: 

Date: 30/06/2025

Co-Supervisor's Name: Mrs. S.S. Thrimahavithana

Signature of Supervisor: 

Date: 30/06/2025

Abstract

Grocery shopping often presents a frustrating experience for customers due to factors such as price discrepancies, inventory inaccuracies, and the lack of personalized assistance. Existing methods, such as store subscriptions and advertisements, are limited in their ability to deliver tailored, timely information. Furthermore, current applications require users to manually browse multiple marketplaces to locate desired products, lacking automated recommendations based on shopping lists. Addressing this gap, this undergraduate project proposes a store recommendation system designed to enhance the shopping experience. The system targets four main objectives: (1) recommending the most cost-effective stores with efficient routes for purchasing a full shopping list, (2) identifying purchase patterns and predicting future shopping needs, (3) optimizing travel paths considering dynamic conditions, and (4) proposing standardized methods for store data collection. To achieve these goals, several algorithms were implemented: greedy heuristics, branch and bound, beam search, and exact optimization methods for store selection; an adaptive genetic algorithm combined with A* search for route planning; and a Singular Value Decomposition (SVD)-based model for personalized item recommendations. Experimental results demonstrated that optimized store suggestion algorithms delivered near-optimal results within acceptable time constraints, and that route planning methods effectively reduced travel time. Challenges related to real-time inventory data acquisition are also discussed, alongside proposals for future refinements to improve data accuracy and recommendation quality. This work highlights the potential of integrated recommendation and optimization systems to significantly streamline the grocery shopping experience and sets a foundation for future real-world deployments

Contents

1	Introduction	1
1.1	Motivation	3
1.2	Scope	4
1.3	Problem Statement	4
1.4	Proposed Solution	4
1.4.1	Included in Scope	4
1.4.2	Excluded from Scope	7
1.5	Aims and objectives of the project	9
2	Literature Review	10
2.1	The Rise of M-Commerce and Location-Based Services	10
2.2	Recommendation Systems	11
2.3	Route Planning Algorithms	13
2.4	Data Collection Methods and Data Standardization	14
2.5	Existing Solutions	17
2.6	Research gap and research questions	20
3	Methodology	22
3.1	Software Development Perspective	22
3.1.1	System Overview	22
3.1.2	Technologies	25
3.1.3	User Interface Designs	28
3.2	Research Perspective	33
4	Implementation	36
4.1	Store Recommendation System Overview	36
4.1.1	Problem Definition	36
4.1.2	Single-Store Mode	37
4.1.3	Multi-Store Mode	38
4.1.4	NP-Hardness of Multi-Store Mode	38
4.1.5	General Algorithmic Approaches	39
4.1.6	Impact of Influence Parameters	40

4.2	Single-Store Mode Implementation	41
4.3	Multi-Store Mode Implementation	43
4.4	Naive Algorithm	43
4.4.1	Time Complexity Analysis	43
4.4.2	Space Complexity Analysis	44
4.4.3	Pros and Cons	45
4.5	Memory-Optimized Algorithm	46
4.5.1	Time Complexity Analysis	47
4.5.2	Space Complexity Analysis	47
4.5.3	Pros and Cons	47
4.6	Filtered Greedy Algorithm	49
4.6.1	Time Complexity Analysis	49
4.6.2	Space Complexity Analysis	50
4.6.3	Pros and Cons	50
4.7	Branch and Bound Algorithm	53
4.7.1	Pipeline	54
4.7.2	Optimization Steps	56
4.7.3	Pros and Cons	57
4.7.4	Limitations and Future Improvements	58
4.8	Beam Search Algorithm	59
4.8.1	Algorithm Description	60
4.8.2	Pipeline	61
4.8.3	Optimization Steps	62
4.8.4	Pros and Cons	63
4.8.5	Limitations and Future Improvements	64
4.9	Integer Linear Programming	66
4.9.1	Algorithm Description	67
4.9.2	Pipeline	68
4.9.3	Optimization Steps	70
4.9.4	Pros and Cons	70
4.9.5	Limitations and Future Improvements	71
4.10	Store Route Planning	73

4.10.1	Problem Definition	73
4.10.2	Methodology	73
4.10.3	A* algorithm	74
4.10.4	Genetic Algorithm	74
4.10.5	Complexity Analysis	75
4.10.6	Pros and Cons	76
4.11	Shopping list recommendation algorithm	78
4.11.1	Data Generation	78
4.11.2	Recommendation Model Training and Evaluation	79
4.11.3	Parameter Exploration	81
4.12	Discount Engine	81
4.13	Best Practices for Standardisation	84
5	Results and Analysis	88
5.1	Evaluation of Route Planning Algorithm	88
5.1.1	Evaluation Methodology	88
5.1.2	Performance Metrics	88
5.1.3	Dataset Generation	89
5.1.4	Parameter Configurations	90
5.1.5	Result and Analysis	90
5.2	Evaluation of Optimisation Algorithms	94
5.2.1	Algorithm Descriptions	94
5.2.2	Comparison	96
5.2.3	Summary of Optimisation algorithms	101
5.3	Evaluation of Recommendation Model	101
5.3.1	Model Performance Hierarchy:	102
5.3.2	Impact of Recommendation Count (N):	105
5.3.3	Influence of Simulation Parameters:	106
5.4	Qualitative Survey Results	110
6	Discussion and Conclusion	112
6.1	Discussion	112
6.2	Conclusion	114

7	Future Directions	117
A	Pseudocodes for Store Recommendation Algorithms	120
A.1	Naive Algorithm	120
A.2	Memory-Optimized Algorithm	124
A.3	Filtered Greedy Algorithm	131
A.4	Branch and Bound Algorithm	136
A.5	Beam Search Algorithm	139
A.6	Integer Linear Programming Algorithm	142
A.6.1	Attempted Implementation	144
B	Psuedocodes for Route Planning Algorithms	147
B.1	A* Algorithm	147
B.2	Genetic Algorithm	150
C	Discount Engine	154
C.1	CFG of Rule Language(DCDQL)	154
C.2	Discount Engine	155
D	Psuedocodes of Data Generation Scripts	164
D.1	Route Planning Algorithm Dataset Generation	164
	References	166

List of Figures

3.1	Initial Software Architecture	23
3.2	Overall Scaled Software Architecture	24
3.3	Request and data flow for the main use case through the scaled architecture	26
3.4	Landing Screen	29
3.5	User creating a new cart	29
3.6	User adding a product item to a cart	30
3.7	User selecting a point on the map	30
3.8	User selecting a route on the map to search using a path	31
3.9	User searching for recommendations	31
3.10	User clicking on a store marker	32
4.1	Vertical waterfall diagram of the algorithm for single-store mode recommendation.	42
4.2	Vertical waterfall diagram of the Parallel Branch and Bound algorithm.	56
4.3	Vertical waterfall diagram of the Beam Search algorithm.	63
4.4	Vertical waterfall diagram of the optimized Branch and Bound algorithm.	66
4.5	Vertical waterfall diagram of the Integer Linear Programming algorithm.	69
4.6	Main Flow Chart of the Route Planning Algorithm	77
4.7	Selected Screenshots of the discount definition interface - set 1	83
4.8	Selected Screenshots of the discount definition interface - set 2	83
4.9	Selected Screenshots of the discount definition interface - set 3	84
5.1	Total distance vs. generations across different datasets	91
5.2	Execution time vs. dataset size	92
5.3	Execution time vs. Generations	92
5.4	Execution time vs. Population	92
5.5	Execution time vs. Crossover	92
5.6	Convergence vs. dataset size	93
5.7	Convergence vs. Population size	93
5.8	Convergence vs. Generation	93
5.9	Convergence vs. Crossover	93
5.10	Recommendation Algorithm Average Durations	98

5.11 Recommendation Algorithm Average Best Scores	99
5.12 Recommendation Algorithm Average Success Rates	99
5.13 Distribution of average f1 score by model	102
5.14 Scores for a normal scenario by SVD	104
5.15 Scores for a normal scenario by SVD++	104
5.16 Impact of number of recommendations (n) on average scores	105
5.17 Average F1-score vs Items per List vs Relevancy Threshold	106
5.18 Average F1-score vs Items per List vs Transactions per User	107
5.19 Average F1-score vs Relevancy Threshold vs List Persistence Threshold	108
5.20 Average F1-score vs Lists per User vs List Persistence Threshold	109

List of Tables

2.1	Comparison of Applications: Inventory and Locator Features	18
2.2	Comparison of Applications: Scalability, Integration, and User Base . .	19
4.1	Store Locator API and data collection Guidelines	87
5.1	Best Route Optimization Results by Dataset (TD - Total Distance, CG - Convergence Generation)	90
5.2	Average Execution Time by Dataset and Configuration	91
5.3	Comparison of Branch & Bound Variants	96
5.4	Comparison of ILP and Beam Search	97
5.5	Performance Timing Metrics for Each Algorithm	98
5.6	Performance Quality Metrics for Each Algorithm	98
5.7	Detailed Statistics for Branch and Bound	99
5.8	Branch and Bound Performance by Number of Items	100
5.9	Evaluation Results of Recommendation Algorithms	100
5.10	Comparison of Data Collection Methods for Inventory Updates	112

1 Introduction

The modern grocery shopping experience presents a significant challenge for many consumers. Busy lifestyles and an abundance of stores and online providers contribute to a time-consuming and often frustrating process. Consider this common scenario: you're rushing home from work when a family member calls requesting groceries. Faced with limited time and information, you choose a random store, potentially missing out on better deals or higher-quality products available elsewhere within a few meters of you or on the same route. This issue arises from the inability to make informed decisions due to lack of relevant information (e.g. product availability and prices).

To tackle this problem vendors often rely on email marketing, subscription-based marketing, and website updates to communicate offers and promotions. However, many customers struggle to keep up with this information overload, leading to missed opportunities. Additionally, these broadcast messages may not be relevant to individual needs and preferences.

This research project proposes to develop a novel software solution designed to improve the overall shopping experience of a user. Imagine being able to query for stores within a particular radius or range to buy a list of products and finding the most cost-efficient (travel distance costs and preferences) stores to buy the items from a list of items. The project aims to provide the necessary information that consumers can use to make informed decisions when it comes to purchasing daily essentials. Most of the existing solutions in modern days would require the user to log in to an application and peruse through their marketplaces to find the products that the user wants. This is time-consuming. Some domain-specific or purpose-specific applications can be found in the real world that would let users enter what they want, along with their quantities, and find the nearby stores, but these are either specific to mall centres, shopping complexes, or specific to particular store chains for the user, which would require the user to be on the same network or a member of a store chain. This research aims to prove, this concept need not be specific to a store chain or be restricted to a shopping mall complex.

Many e-commerce applications related to location-based systems are gaining more and more attention. This project is set to explore more into the location-based systems and location-based services, in this domain to accomplish the location-based filter-

ing of stores, distance calculations, nearby store suggestions, and as an influencing factor of recommendations. This project also explores recommendation systems with multi-dimensional considerations, along with route recommendation and planning algorithms.

1.1 Motivation

The challenges outlined previously resonate with many grocery shoppers. Finding accurate product and inventory information for specific stores, let alone identifying the cost-effective store for a shopping list, remains a persistent barrier. These difficulties arise due to the lack of a centralized platform.

As mentioned, this research aims to expand the concept of search rather than looking or perusing. This research seeks to alleviate the burden of looking for products on their own and spending a lot of time. Due to this, the amount of unnecessary expenditures can also be reduced significantly due to the omission of clickbait. How many times have customers bought some product items that they never intended to buy while searching for some products? Very often. Users can search for what they want without unnecessary distractions.

Furthermore, the past few years has witnessed significant price fluctuations in Sri Lanka, particularly for essential grocery items. These fluctuations created confusion and frustration for customers, sometimes leading to disputes with store owners. Even existing online platforms and data sources often suffer from infrequent updates, making them unreliable and frequently leading to customer dissatisfaction.

These issues – the inconvenience of information gathering and the overwhelming, impersonal nature of store subscriptions – inspired our desire to develop a solution. Customers are overwhelmed with irrelevant promotional emails, leading to subscription fatigue and missed opportunities for valuable savings on everyday purchases. Their busy lifestyles often prevent them from actively seeking out the best deals on individual store websites.

The current landscape lacks a comprehensive solution that streamlines the grocery shopping experience for both consumers and stores. This research project aims to bridge this gap by proposing a novel solution that addresses these needs.

1.2 Scope

This research project proposes the development of an application designed to provide accurate and helpful information when purchasing groceries to make informed decisions. The scope of this project is oriented around three broad areas: recommendations and combinatorial optimization problems for time-sensitive systems and data aggregation methods, which will be narrowed down to three sub-tasks to achieve the research aim within the duration of this research project.

1.3 Problem Statement

Grocery shopping can be a time-consuming and often frustrating task. Consumers frequently face challenges such as inefficient information on prices and offers, price discrepancies across stores, limited information on product availability, and a lack of insightful personal guidance to guide purchasing decisions.

1.4 Proposed Solution

The proposed solution revolves around three main items. Getting the data to the system in a reliable, timely manner. Providing recommendations by solving a combinatorial problem (users will enter a list of items system will fetch the nearby stores and their product information, and will tell the customer the most cost-effective store or stores to purchase the list of items from) , and personalization (giving notifications and recommendations on frequently purchased items). The final outcome of this would be a user-centric mobile application that offers the following key functionalities.

1.4.1 Included in Scope

Main functions associated with the user-side (customers) are,

- Registering to our services (which will create a profile for the user).
- Login to the platform.
- Adding user preferences (which will be helpful in the recommendation stage).

The following are some of the preferences users can preset to guide the recommendation process:

- Search radius.
 - Payment method preferences (cash or digital payment methods) and preferred digital payment type.
 - Input a list of grocery items to start the search.
 - Save prior inputs (save the lists of grocery items) and the ability to view them at any time as needed, and search for them.
 - Update any of the existing lists, save the list as a new list or save the updated version of the grocery list.
 - Start searching for stores using a new input or an existing grocery list.
 - View previous recommendations for a grocery list.
 - Users can search for recommendations for two different geographical properties.
 - * Point - a single location.
 - * Path - a line-string with a starting location and a destination.
 - Users can search for store recommendations using two different modes.
 - * Buy all the products mentioned in the list at a single store.
 - * Buy all the products collectively from different/multiple stores.
- Note: Either way user can request several recommendations, from which the user can select a recommendation.

Main system responsibilities associated with the user inputs are,

- Creating the user profiles.
- Generate store recommendations.
- Display overall cost calculations of the recommended stores for a given list of items.
- After selecting a recommendation, give clear directions using integrated mapping functionalities to the store(s).

- Save previous recommendations, save user shopping lists.

Main functions associated with store-side are,

- Registering to the platform.
- Login to the platform.
- Managing (editing different aspects of the profile such as username, and etc.) the store profile (which was created when the store was registered).
- Registering a data provider to the platform (Note: providing accurate data regarding products, prices, and discounts is the responsibility of the store owners.).

Main system outputs related to store-side inputs are,

- Creating the store profiles.
- Aggregating the necessary data from the registered data providers for recommendations.
- Maintain store-product information.

Responsibilities of the system,

- User data and privacy protection through robust storage practices and a comprehensive privacy policy.
- Unbiased store recommendations - recommendations must be purely based on cost and distance metrics; store-related data should not be used as influential factors.
- Providing accurate recommendations promptly (within a reasonable time; currently the metric is set to 4 seconds for very high data volumes).

The project scope encompasses two primary categories:

- Core Functionalities:
 - User profile management.
 - Grocery list creation, editing, deletion, and storage.
 - Store locator with map integration and directions
 - Price comparison engine for nearby stores.
 - Secure data storage and privacy policy adherence.
- Functionalities Requiring Further Research:
 - Developing a recommendation algorithm to analyze different metrics between stores to recommend cost-effective recommendations and solve combinatorial set cover problems for high data volumes.
 - Establishing a standardized methodology for grocery stores to publish inventory information through API integration.
 - Utilizing content-based and collaborative filtering techniques to personalize recommendations and forecasting of possible purchases based on user profiles and vendor-provided details.
 - Route and trajectory planning component for intelligent route recommendations and picking the stores with the least travel distance to minimize traveling costs.

1.4.2 Excluded from Scope

The following are the components that are not included in the scope for the duration of this research project.

- Application will not provide users with chat-bot recommendations based on a list of items that users want to purchase.
- Integrations with other platforms that provide online purchases and other services that are related to online purchases will not be done.

- Product management functionalities or applications to stores (Inventory management) will not be provided.
- Even though there are multiple methods discussed throughout the document, about data extraction, only the API based method will be encouraged.
- Personalized Recommendation system will not use contextual hints, thus will have no understanding of contextual information.
- System will not consider factors that contribute to latency, such as queues and waiting times, as the primary focus is on recommendations, while leaving room for more influential factors as such.

The application will not provide any functionalities related to social media or integration with social media platforms. The initial outcome of this research will only focus on store recommendations, route and trajectory management and planning.

In the initial increment of the solution, data collection from stores will happen through APIs provided by the vendors and it is decided that the vendors must provide accurate information related to the prices and quantities of the products, and it is also the responsibility of the vendors to update the information periodically based on the rate of their sales as inventory management is beyond the scope of our application.

The final outcome of this research will be tailored to fit the requirements of consumers, retail application guidelines, and the route system of Sri Lanka while emphasizing the possibility of adaptation to other regions. The main focus will be in the context of Sri Lanka.

1.5 Aims and objectives of the project

The aim of this project is to successfully cover the previously agreed scope and provide answers using existing technologies through software engineering principles in the domain that was discussed thus far. The aim is to provide a software solution that can optimize the overall grocery shopping experience. The following can be mentioned as the object of our project:

- Develop and test a recommender system that can recommend the most suitable stores for an individual to buy grocery products (multi-store recommendations or single-store recommendations).
- Develop and test a route and trajectory recommender system to reach the stores efficiently and cost-effectively.
- Develop and test a recommender system that can deliver recommendations about the grocery products that a particular individual might be interested in based on user data and prior interactions.
- Develop a methodology to get inventory data from the stores promptly, which will provide the blueprint for getting information from stores.
 - Note: Due to timeline considerations, data-related limitations and hardware limitations, and survey results, the research deviated from the original plan for achieving this particular research aim (more information will be provided in later sections).

2 Literature Review

The evolution of m-commerce and the crucial role of recommendation systems are explored in this literature review, with a particular focus on how location-based services enhance personalized user experiences. Various recommendation techniques are explored, and existing gaps in time-sensitive, location-aware applications, especially within the context of grocery shopping, are addressed.

2.1 The Rise of M-Commerce and Location-Based Services

Since the inception of smartphones, e-commerce has entered a new trajectory, which has rise of m-commerce and various innovations in the m-commerce domain. M-commerce has enabled the opportunity to incorporate new dimensions of information about a user, such as their geographical location, context capturing (understanding the environment of the user), and many other attributes to provide more sophisticated services and personalised services to the users more conveniently (Safavi, Jalali, and Houshmand 2022). Modern day, recommendation systems are used by almost all digital service providers to improve the user experience through personalised recommendations, which will further contribute to the retention of customers. When it comes to time-sensitive systems, however, there is a lack of software that uses such technologies to improve customer experience.

When it comes to the m-commerce domain, with the technological advances, it's easy to acquire the user's geographical location, which is very important in the domain of discourse. A common example of some systems that take advantage of such smart devices would be Uber and PickMe taxi services. And also, when you visit one of those applications, you will often see that they are not limited to a single discipline. For example, Uber Eats, and PickMe Foods, provide services other than transportation services through those applications. Thus, the extent to which such location-based applications(including point-of-interest applications) can be used is large and complicated services can be built around such technologies.

2.2 Recommendation Systems

Recommendation systems have a famous reputation in many domains, especially in the mobile-based marketing domain(m-commerce). As many people have smartphones with them, it's easy to reach a wider audience. Vendors can use different types of recommendation systems to provide recommendations for their clients. For example, a vendor can use an email system to send product information. However, the preferences of each customer will vary. Due to this, most of the information that people are getting is overwhelming. Since the introduction of more intelligent recommendation systems, people have tried to provide more personalised recommendations for their audience (Safavi, Jalali, and Houshmand 2022). Through data analysis techniques, vendors might want to find out which products the user will likely buy based on the user's previous interactions. Such recommendation systems consider many disciplines and dimensions when it comes to recommendations. Incorporation of such information varies based on the techniques used to design such recommendation systems.

Despite the widespread adoption of recommendation systems, most do not incorporate location as a variable in content-based or collaborative filtering techniques. The emergence of social media platforms, such as Facebook, has highlighted the demand for "point-of-interest" (POI) systems. A prominent example is hotel recommendation systems, which suggest accommodations for travelers. System designers have access to a variety of recommendation algorithms to develop such systems. One effective approach involves the integration of two distinct components tailored to different recommendation types. For instance, an online recommender system can be implemented to provide real-time suggestions, such as identifying the most suitable location for a user to purchase a beverage while traveling. Additionally, users may seek recommendations informed by their prior interactions and usage patterns. However, conducting extensive data analysis on large datasets to deliver such insights in a timely manner poses challenges, potentially resulting in suboptimal user experiences. To address this, an offline recommender system can be introduced to process substantial data volumes and generate user profiles based on their interactions. This dual-system architecture enhances overall performance by enabling the online recommender to deliver prompt responses while the offline system provides in-depth analysis. The subsequent section will examine the techniques utilized in various recommender systems in greater detail.

Several types of recommendation systems have been studied and the main types are identified as: Content-based recommendation systems, Collaborative filtering-based recommendation systems, Hybrid recommendation systems, Context-aware recommendation systems and Knowledge-based recommendation systems. Content-based recommendation systems use a user's given set of preferences and the properties existing on an item to check whether it is suitable to be recommended to a user. This can be achieved in a number of methods such as the nearest neighbour method, decision trees, relevance feedback calculation and more, as explained by Pazzani and Billsus 2007. Collaborative filtering is a concept based on how users' behaviour repeats and derives recommendations from that. They are classified into model-based collaborative filtering (Using various machine learning models to predict recommendations) and memory-based collaborative filtering (Using collected previous user behavior data to generate recommendations). Hybrid recommendation systems combine more than one recommendation system to refine recommendation output by reducing drawbacks of each recommendation system with the help of other recommendation systems. They can be combined in a few ways like weighted hybridization, switching hybridization, feature combination and more hybridizing techniques as explained by Burke 2002. Context aware recommendation systems have information about the context (information describing the environment the system is deployed in) which is obtained by the system via explicit inputs, implicit automatic capture and inference as explained by Verbert et al. 2012. The contextual information can be used to filter recommendations from another recommendation technique before or after it is applied and the recommendation technique can use the data directly in it. These approaches are called contextual pre-filtering, contextual post-filtering and contextual modeling (Verbert et al. 2012). As described by Burke 2002, Knowledge based recommendation systems are based more on expert knowledge related to the domain the recommendation system is being made for rather than a huge pool of data that is used to derive recommendations from. These systems consist of two core components which are the knowledge base and the inference engine. The knowledge base stores the domain specific knowledge and the inference engine uses that knowledge to generate recommendations (Burke 2000). A combination of features from the above mentioned recommendation systems should aid in creating a recommendation system that fits any general use case.

2.3 Route Planning Algorithms

Finding the most optimal route based on several criteria is quite a challenging problem. This algorithm addresses the challenge of selecting the optimal combination from pre-generated store combinations by the store recommendation system while accounting for the travel cost (total distance and traveling time) in a traffic-aware context. The algorithm incorporates real-time traffic data to ensure accurate traveling time estimations.

The A* algorithm which is heuristic-based pathfinding algorithm was introduced by Hart, Nilsson, and Raphael 1968, It is most suitable for grid-based environments. Its application for Automated Guided Vehicle, as demonstrated by Tang et al. 2021. Because this is mainly designed for static environments, does not primarily support dynamic conditions such as traffic conditions.

The Genetic algorithm approach is a bio-inspired technique that evolves the generation based on multiple objectives. Using evolutionary processes including selection, crossover, mutation, and elitism, the Genetic Algorithm (GA) looks through the set of store combinations and their potential visiting orders to find the combination and sequence that minimizes travel costs (distance and time) (Srinivas and Patnaik 1994). The adaptive genetic algorithm with the A* algorithm was developed to address the problem of finding optimal store combinations and the optimal store sequence based on several factors. The algorithm has two main components.

- Global optimization with Genetic Algorithm

Starting with pre-generated store combinations, it operates the evolutionary process through selection, fitness, crossover, mutation, and elitism steps to evolve the solutions.

- Local pathfinding with A* Algorithm

This algorithm computes the optimal route for the given store order using a heuristic approach that leverages distance and real-time traffic information.

Future enhancements to the Adaptive Genetic Algorithm with A* Algorithm will focus on incorporating additional factors.

2.4 Data Collection Methods and Data Standardization

Every recommendation system requires data. In the context of the proposed systems, it is imperative to acquire knowledge about the inventory data of stores within a range specified by user preferences. This necessitates a method for retrieving data from stores both promptly and with high accuracy. Currently, the predominant method for obtaining such data is through Application Programming Interfaces (APIs) (Murphy et al. 2018). However, this approach demands predefined data formats. The system must be capable of interpreting and utilizing the data received effectively. A predefined data mapping is essential to process the incoming data accurately. Furthermore, the real-time aspect of the system, including the frequency of data updates, constitutes a critical factor that must be considered. This element is of significant importance in the design process. The provision of standardized Application Programming Interfaces (APIs) offers an accessible and efficient solution for application designers in this field, ultimately fostering greater innovation (*Data Exchange Mechanisms and Considerations* 2020).

Enterprise applications can use different methods to feed data to receivers. The methods used depend on different aspects of the organisation. Some of the common criteria are as follows,

- Data Characteristics
- Data Environment Characteristics:
 - Data Security
 - Frequency of Usage
 - Data Versioning
- Organisational Considerations
- Scope Constraints: Budget and time constraints
- Consumer Characteristics

Data exchange patterns are usually constructed based on three main aspects as mentioned by *Data Exchange Mechanisms and Considerations* 2020.

- An Architectural Pattern
- A Data Format
- A Communication Protocol

Based on the requirements of the projects, such aspects can be determined. Systems with a higher frequency of changes and requests should be able to scale without much effort and should have near-zero downtime. Another aspect to consider is the timeliness of the data, whether synchronous or asynchronous methods should be preferred to maintain consistency. Research suggests efficient product data collection uses standardised identifiers like GTINs (Global Trade Item Numbers) (*Global Trade Item Number* 2025a) and supplier APIs for accuracy. It seems likely real-time stock updates rely on POS (Point of Sales) integration, barcode/RFID scanning, and cloud systems for speed and precision through periodical updates. (ST and Perishable 2025) Datascan provides tools specialised in inventory tracking scanners, inventory tracking system (software), and RFID based method with enhanced data analytics, to carefully manage and track inventory information. These platforms can be coupled with a store locator application to keep inventory information updated. Any of these systems require hardware support, such as RFID (barcode) scanners or sensors, to make the process much quicker. Implementing these barcode or RFID based methods have its own advantages, leading to smart IDS and increasing traceability (*GS1* 2025). Forbes has provided some guidelines on retail inventory management best practices as well (Baluch 2022). One of the main highlights can be found in reducing dead stock, which leads to cost savings and increased profits by re-stocking what's relevant and is crucial in understanding the trends in the market.

To collect product information efficiently with high accuracy, use standardised identifiers like GTINs, which ensure unique product identification and high quality. Retrieve data from suppliers via APIs or data feeds, such as those supported by GS1's (*Global Trade Item Number* 2025b) Global Data Synchronisation Network (GS1 Standards). For retail chains, maintaining a central database to ensure consistency and integrating with POS and barcode systems for tracking can be found as widely used and famous methods. Even though data collection methods are a possibility, defining standards (consistent data formats) is a very time-consuming process and takes a lot of effort

to adapt. Most of these projects are led by the government sector as a part of their initiative to improve data governance and standardisation. As mentioned, these solutions are used by large-scale operations, but the core idea seems to remain adaptable even for small-scale projects with proper devices and software tools.

Further more research provides valuable insights into the standardization process, particularly for large-scale applications, from a theoretical perspective, In the paper (Gal and Rubinfeld 2018), “Data Standardization,” Michal Gal and Daniel L. Rubinfeld argue that standardization enhances data portability and interoperability, improving machine learning and data-driven decisions (Data Standardization). However, they highlight costs like privacy risks and cybersecurity concerns. They suggest governmental facilitation is necessary for cross-industry or inter-temporal data synergies, using regulatory methods like mandates or incentives. This is relevant for statewide store locators, where diverse stakeholders require coordinated standards. These studies suggest that standardization requires balancing benefits (e.g., interoperability) with challenges (e.g., privacy), and a collaborative, governed approach is critical for success. Some real-world examples can be mentioned as follows,

- National Information Exchange Model (*NIEMOpen For Government / An OASIS Open Project* 2025):
 - NIEM is a community-driven framework led by the United States government for standardizing information exchange across federal, state, and local agencies, used in areas like public safety and child welfare.
 - Their standardisation process uses a common vocabulary with reusable data components (e.g., person, address) defined in XML schemas.
 - NIEM supports standardized XML-based APIs, with conformance rules ensuring consistency.
 - NIEM provides guidelines for data collection, ensuring semantic consistency across organizations. It uses a standardized lifecycle for developing exchanges.
- General Transit Feed Specification (GTFS)(*General Transit Feed Specification* 2025):

- GTFS is an open standard for public transportation schedules, used by thousands of transit agencies worldwide, including integration with Google Maps.
- GTFS defines a set of CSV files (e.g., `agency.txt`, `routes.txt`) with specific fields, packaged in a ZIP file. The format is simple yet comprehensive, covering routes, stops, and schedules.
- GTFS focuses on data files; its real-time extension (GTFS Realtime) uses Protocol Buffers (*Protocol Buffers* 2025) for API-based updates, ensuring standardised real-time data feeds.
- Information collection is handled through transit agencies, which collect data in the GTFS format, often using software tools to generate compliant feeds. The standard’s simplicity encourages adoption.
- GTFS’s straightforward format and community support make it a model for standardising product and inventory data in a store locator, especially for smaller retailers.

By looking at these two cases, it is understood that the process of standardisation and the data formats being used differ from process to process and ease of use influences the use of these methodologies as they are easier to adapt and get used to.

2.5 Existing Solutions

Several existing systems exhibit similarities to the functionalities targeted by the proposed grocery shopping recommendation system. For instance, Target (Target 2025) offers a store locator service to identify stores within its chain that stock specific items, and Walmart provides a comparable feature (Walmart 2025). However, these applications are limited to their respective store chains, restricting their scope. The following applications, summarized below, offer related functionalities but with distinct objectives.

- Locally (*Locally* 2025): Collects real-time inventory from over 55,000 stores globally, guiding shoppers to nearby purchase locations.

- NewStore (*NewStore - A Unified Commerce Platform for Global Brands* 2025): Offers a comprehensive real-time inventory management system integrated with retail operations.
- Lightspeed Retail (Lightspeed 2023): A POS system with inventory management software that syncs accurate inventory data in real-time.
- Storemapper (*Storemapper: Customizable Store Locator App and Software* 2025): Provides a store locator app that helps connect customers to stores, potentially including inventory checks.

The following tables (2.1 and 2.2) compare these applications based on key features and attributes.

Table 2.1: Comparison of Applications: Inventory and Locator Features

Application	Real-Time Inventory	Store Locator
Locally	Yes, from 55,000+ stores	Yes, guides to buy locations
NewStore	Yes, with ATP and RFID	Indirect, via retail ops
Storemapper	Possible, not explicit	Yes, customizable
Lightspeed Retail	Yes, real-time sync	Indirect, via POS

The applications under consideration—Locally, NewStore, Storemapper, and Lightspeed Retail—each offer unique solutions in the realm of retail technology, particularly focusing on inventory management and store location services. Locally is a platform that connects shoppers with over 55,000 stores globally, providing real-time inventory data and guiding users to nearby purchase locations (*Locally* 2025). NewStore offers a comprehensive real-time inventory management system integrated with retail operations, utilizing advanced technologies such as Available to Promise (ATP) and Radio-Frequency Identification (RFID) (*NewStore - A Unified Commerce Platform for Global Brands* 2025). Storemapper provides a customizable store locator app that helps businesses connect customers to their physical stores, with potential inventory check capabilities (*Storemapper: Customizable Store Locator App and Software* 2025). Lightspeed Retail is a point-of-sale (POS) system that includes inventory management software, ensuring accurate and real-time synchronization of inventory data across multiple locations (Lightspeed 2023).

Table 2.1 compares the applications based on their real-time inventory management and store locator features. For real-time inventory, Locally and Lightspeed Retail explicitly provide this functionality, with Locally sourcing data from a vast network of stores and Lightspeed Retail offering seamless synchronization through its POS system. NewStore also supports real-time inventory but emphasizes integration with retail operations using ATP and RFID technologies. Storemapper’s capability in this area is less explicit, suggesting that while it may offer some form of inventory checking, it is not a primary feature. Regarding store locators, both Locally and Storemapper directly provide this service, with Locally guiding shoppers to buy locations and Storemapper offering customizable maps for businesses. NewStore and Lightspeed Retail, on the other hand, provide store locator functionalities indirectly through their broader retail and POS systems, respectively.

Table 2.2: Comparison of Applications: Scalability, Integration, and User Base

Application	Scalability	Integration	User Base
Locally	High, global reach	Dashboards, API/BI tools	60M+ shoppers, 1,000+ brands
NewStore	High, multi-channel	ERP, Ecomm, POS	Not specified
Storemapper	Medium, business-focused	WooCommerce, etc.	Thousands of businesses
Lightspeed Retail	High, multi-location	POS, e-commerce	Small to medium businesses

Table 2.2 evaluates the applications on scalability, integration capabilities, and user base. Locally demonstrates high scalability with a global reach, integrating with dashboards and API/BI tools, and boasts a user base of over 60 million shoppers and 1,000+ brands. NewStore also exhibits high scalability, particularly in multi-channel retail environments, with integrations into Enterprise Resource Planning (ERP), e-commerce, and POS systems; however, its user base is not specified. Storemapper is characterized by medium scalability, focusing on individual businesses and integrating with platforms like WooCommerce, serving thousands of businesses. Lightspeed Retail is highly scalable for multi-location operations, integrating with POS and e-commerce systems, and is tailored for small to medium-sized businesses.

In summary, the literature review explores the evolution of mobile commerce (m-commerce), driven by smartphone adoption, which has enabled location-based services

to enhance personalized user experiences. It highlights the pivotal role of recommendation systems—including content-based, collaborative filtering, hybrid, context-aware, and knowledge-based approaches—in improving customer engagement, though many systems underutilize location data. Route planning algorithms like A* and Genetic Algorithms (GA) are examined for optimizing store visit sequences, with a focus on minimizing travel costs in traffic-aware contexts. The review also addresses data standardization challenges, emphasizing the need for real-time inventory data through APIs, standardized identifiers (e.g., GTINs), and integration with POS systems. Existing retail solutions—such as Locally, NewStore, Storemapper, and Lightspeed Retail—are compared, revealing their strengths in inventory management and store locators but also gaps in unified, time-sensitive grocery shopping systems. These insights underscore the potential for a new system that integrates real-time inventory, location-based recommendations, and route planning, addressing the limitations of current solutions.

2.6 Research gap and research questions

While numerous recommender systems exist to personalise online shopping experiences, a significant gap persists in optimising in-person grocery shopping. This research project aims to address this gap by developing a mobile application that integrates real-time price comparison, route and trajectory planning, and personalised recommendations.

The following research questions delve deeper into the core functionalities of the proposed application:

- How can recommender algorithms be devised and optimized to deliver store recommendations considering multi-dimensional data for grocery shoppers?
- What standardized data format and API integration approach can be adopted by grocery stores to facilitate the efficient exchange of real-time inventory information with a recommendation system, ensuring data accuracy and system scalability?
- What combination of content-based filtering and collaborative filtering techniques can be most effectively utilized within a grocery shopping application’s

recommendation system to deliver highly personalized product suggestions that cater to individual user preferences and evolving shopping habits?

- How can path-finding algorithms be integrated with a grocery shopping application to suggest the most efficient route for users to navigate to the recommended store?

3 Methodology

Methodology and the design of the project are divided based on two perspectives. The software development perspective will offer insights into the development method, technologies used to implement different components of the system, and the high-level software architectures tried and currently in use. The research perspective will offer insights into the research-related components, surveys, technologies used to conduct experiments, and how progress was recorded.

3.1 Software Development Perspective

The project encompasses a defined set of high-level requirements and functionalities, as established in the Subsection 1.4.1, to be implemented in the initial software increment. Each requirement integrates a research component. For instance, the multi-store recommendation mode involves determining sets of stores from which a user can collectively acquire a specified list of items. This process corresponds to the set cover problem, recognized as NP-hard in computational complexity. To address this intricacy, the software development process was designed and executed according to an iterative and incremental model. Here is why,

- Incrementally build the application: Start with a simple version, then add advanced techniques (e.g., optimizations, caching, etc.) in later increments.
- Iteratively refine it: Test each version, analyze weaknesses, and improve it in the next cycle.
- Use of experimental feedback to guide the development and give a clear idea as to what is being implemented.

This process can also increase the modularity of the system due to the incremental development (development of a single feature at a time). Figure (3.2) demonstrates the high-level design of the system.

3.1.1 System Overview

Initial increments of the system adopted a monolithic architecture, Figure (3.1), and as the system and the features grew, main services were identified and the system was

partitioned into the following main components: user-side functionality management (user-service), store-side functionality management (store-service), recommendation model, solver service, notification service, product service and the route recommending service. To increase the performance of each server, a caching service (Redis), an OSRM (Open Street Routing Machine) service for computing real-world traveling distances, and messaging queues were used. These are implemented as separate services and will communicate with each other using message-passing protocols such as MQTT (Message Queuing Telemetry Transport protocol), gRPC (Google Remote Procedure Calls) and REST (Representational State Transfer) for HTTP-based communications. This also provides us with the capability to change the suitable communication protocols between different components with minimal effort. Figure (3.2) demonstrates this scaled version. The internal structure of the services will be discussed in detail in the coming sections.

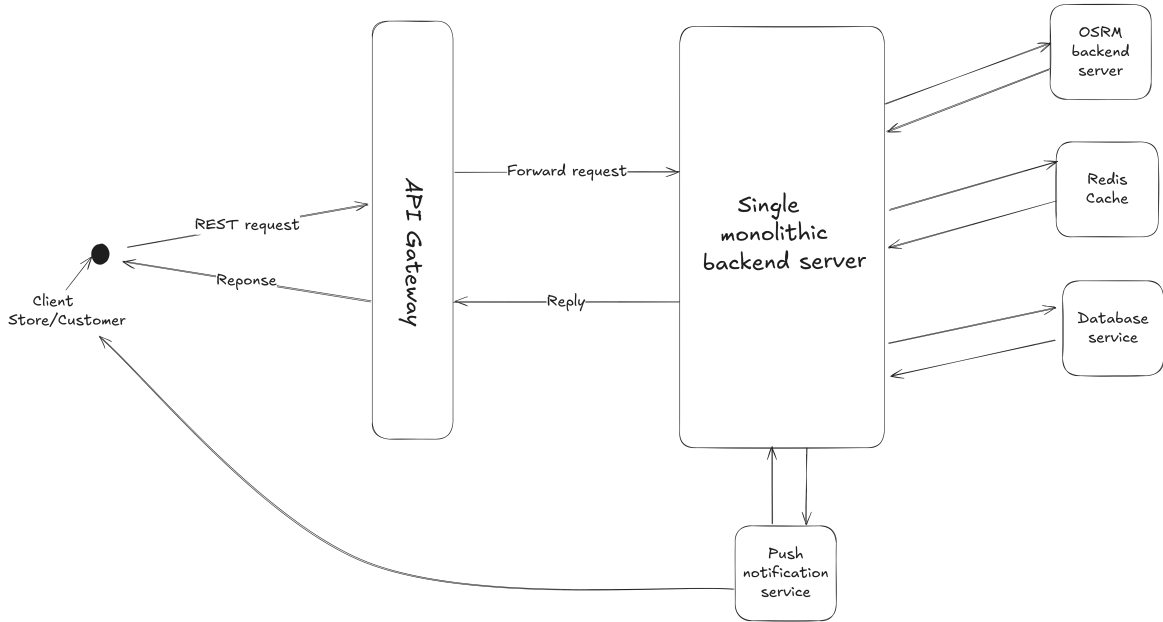
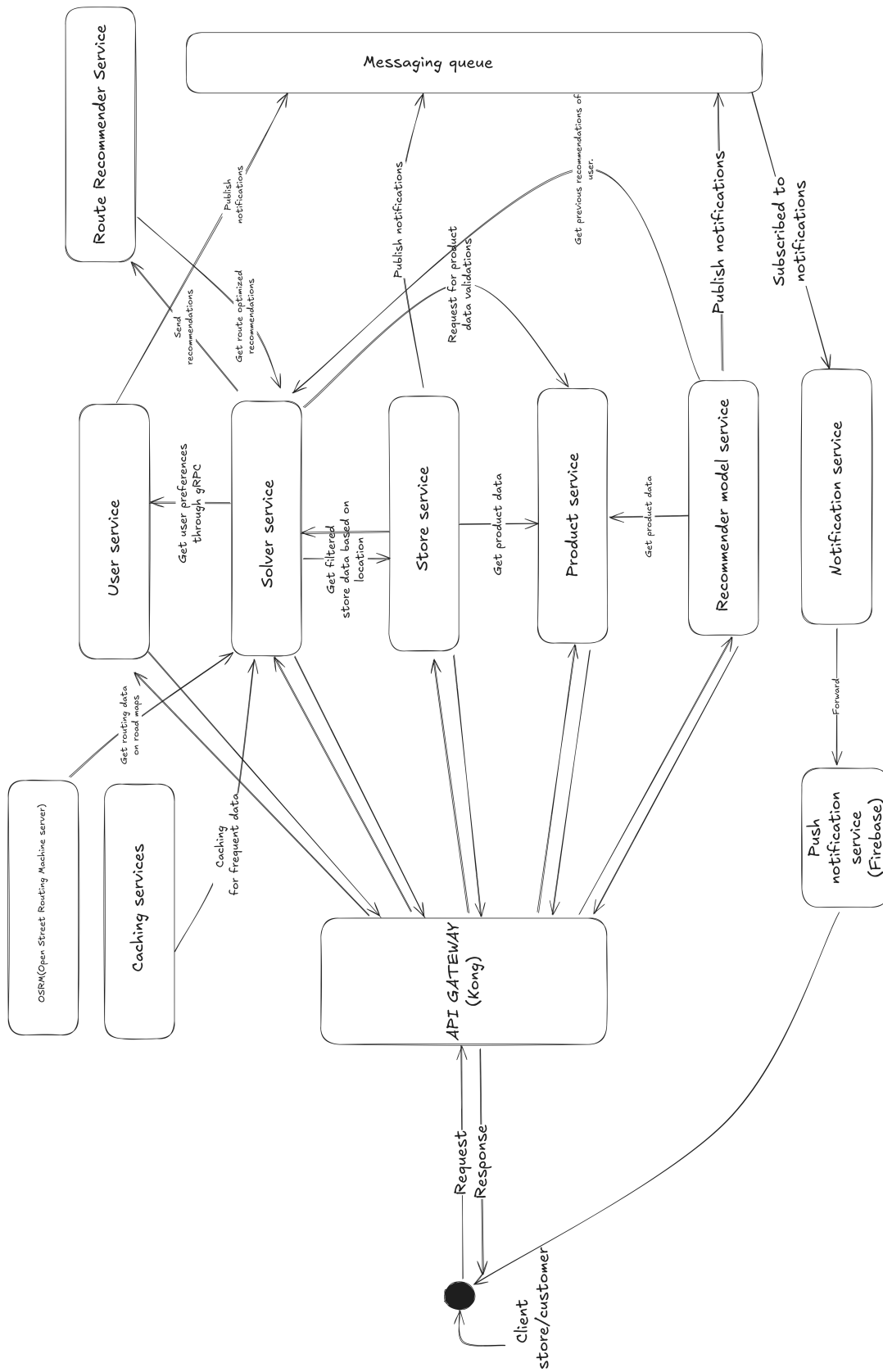


Figure 3.1: Initial Software Architecture

The figure (3.3) depicts the request flow pipeline using an end-to-end request flow through the system. The main functionality of determining which stores to buy products from is considered the main flow. In brief, the user will send a payload with the items, location coordinates of the user, radius, and the recommendation mode (single-store or multi-store recommendations). The API gateway then validates the tokens and, if valid, forwards the request to get the user. If the user exists, the user is



NOTE: Each service has a separate database service to keep their respective data and the inter server communication happens through gRPC and external (out-side worlds) communication happens through REST communication method. For brevity those connections are excluded.

Figure 3.2: Overall Scaled Software Architecture

authenticated and access is granted, and then the solver service is going to validate the list of product items. If valid, it goes to the store service to get the nearby stores and their store-product information, such as unit price, stock and so on. Next, the solver fetches the user preferences as well. If there are enough stores, the solver will run the solving algorithm on the problem at hand. While doing so, the solver service will use the Redis cache server and the OSRM server. And after running the algorithm, it will return the reply.

3.1.2 Technologies

In the development process, the following technologies were used. The reasoning behind the design choices and reasoning will be discussed in later sections.

- Version control and collaboration: Git and GitHub.
- Project management tools: GitHub projects and Trello.
- Operating system: Windows, Debian, and Fedora.
- API request and validation tools: Postman.
- User interfaces and frontend:
 - Web version: Vuejs3, React Native (compiled to web).
 - Mobile: React-native.
- Map functionality: Open Street Maps services (OSM APIs for getting location or path data as GeoJson), OSRM (Open Street Routing Machine, for calculating travelling distances, for walking, and driving, LeafletJs, Turfjs for highlighting the paths or ranges, Google Maps API for frontend map rendering and route displaying.
- Data set generation: Open Street Maps location data APIs, Geofabrik geospatial data, and open food facts for generating product data.
- Programming languages: Elixir (attempted), Golang, JavaScript, TypeScript, Python.

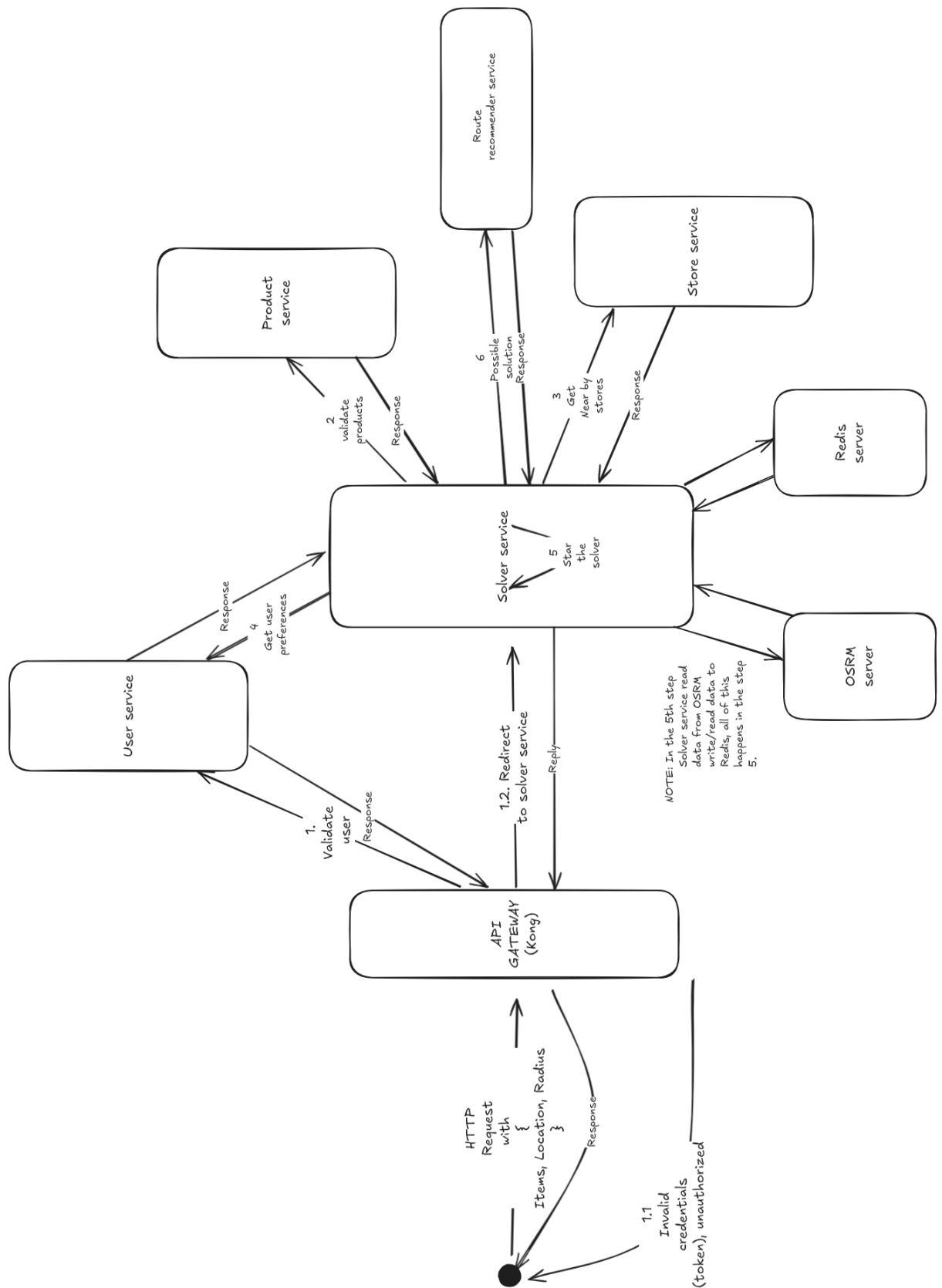


Figure 3.3: Request and data flow for the main use case through the scaled architecture

- Frameworks: NestJs, React-Native, Vue, no backend framework was used other than the Golang standard library for Go-based implementations, Flask, Phoenix (attempted).
- API gateways, server configuration tools, reverse-proxies, live servers: Kong API gateway, Nginx, ngrok.
- Database and memory caching servers: Postgresql, Redis.
- Messaging queues: Rabbitmq server.

The technologies selected for the store recommendation and route planning system were strategically chosen to ensure scalability, efficiency, and a seamless user experience in a real-time, location-based application. Git and GitHub were adopted for their robust version control and collaboration features, facilitating effective team coordination across development phases. Project management was streamlined using GitHub Projects and Trello to organize tasks and track progress efficiently. The system was developed to be cross-platform, supporting Windows, Debian, and Fedora, ensuring compatibility across diverse environments. Postman was utilized for thorough API testing and validation, critical for reliable data exchange. For user interfaces, React Native was selected for its ability to deliver consistent experiences across web and mobile platforms, complemented by Vuejs3 for specific web components due to its reactive framework. Mapping functionalities leveraged Open Street Maps (OSM) and Open Source Routing Machine (OSRM) for cost-effective, accurate location and routing data, enhanced by LeafletJs and Turfjs for interactive geospatial visualization, and Google Maps API for advanced frontend rendering. Data generation relied on open sources like OSM, Geofabrik, and Open Food Facts to minimize costs while accessing comprehensive datasets. Golang was chosen for its concurrency features and extensive standard library, enabling efficient backend development without additional frameworks, while NestJs and Flask supported scalable server-side and Python-based services, respectively. API management was handled through Kong, Nginx, and ngrok to ensure secure and efficient traffic routing. PostgreSQL and Redis were employed for robust data storage and high-performance caching, respectively, while RabbitMQ facilitated asynchronous operations through message queuing, enhancing system decoupling and responsiveness. These technologies collectively address the system's require-

ments for real-time data processing, location-based services, and high performance, ensuring a scalable and reliable solution.

Under the Section 4, several technological concerns related to the backend services are addressed to elucidate the design decisions undertaken. Initially, the project adopted the Elixir programming language and the Phoenix framework as the primary language and backend framework for implementation. However, shortly after the project commenced, it became evident that sufficient support for certain required tasks was lacking, as some necessary libraries were unavailable. Consequently, a shift to the Go programming language was deemed necessary. Go was selected due to its robust concurrency features and comprehensive standard library support, which eliminated the need for an additional backend framework. The out-of-the-box concurrency mechanisms provided by Go facilitated the parallelization of the algorithms detailed extensively in this section. Docker containers were utilized to ensure consistency across both development and deployment environments. As noted in the methodology section, this approach simplified the scaling of the algorithms. Initially, the server was designed as a monolithic architecture; however, it was later restructured into a services-based framework, allowing for the independent scalability of individual components.

3.1.3 User Interface Designs

Under this section, designs used for the mobile application interface will be included. They are implemented as is, and most of them are taken from a hosted working system. This will be an overview to give the reader an idea of the interface of the final outcome.

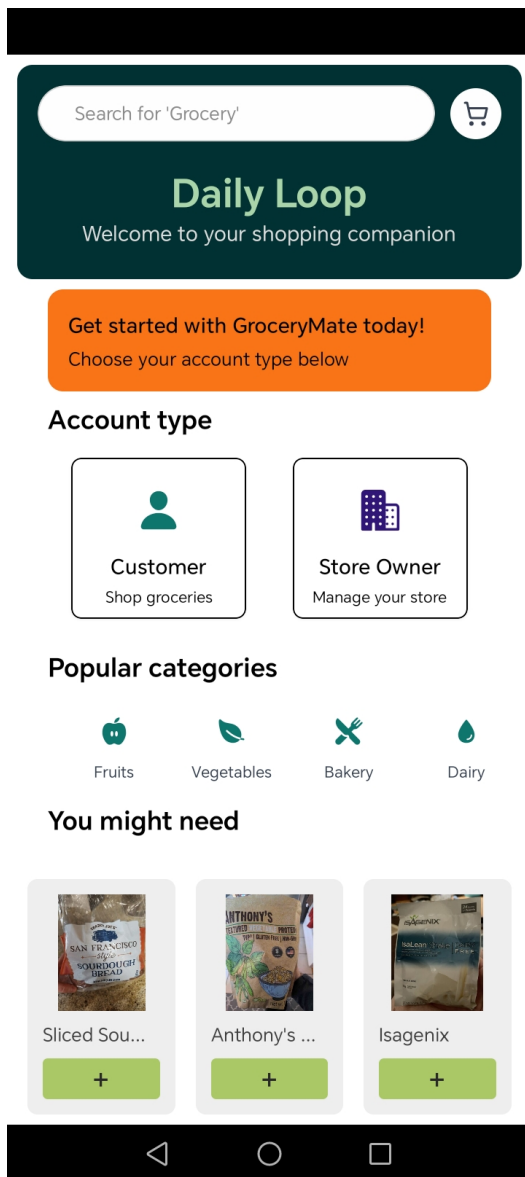


Figure 3.4: Landing Screen

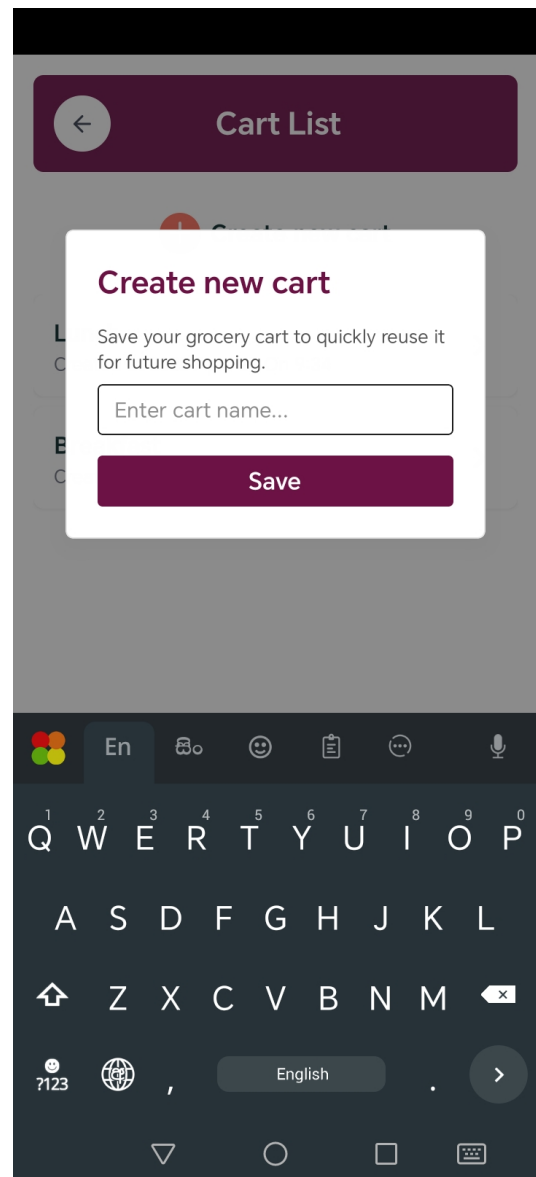


Figure 3.5: User creating a new cart

The very first screen that the users would see upon entering the application is shown by figure 3.4). By selecting appropriate option the user can proceed either as a "Customer" or a "Store owner", from there on wards. Figure (3.4) shows how a customer can create a new shopping cart.

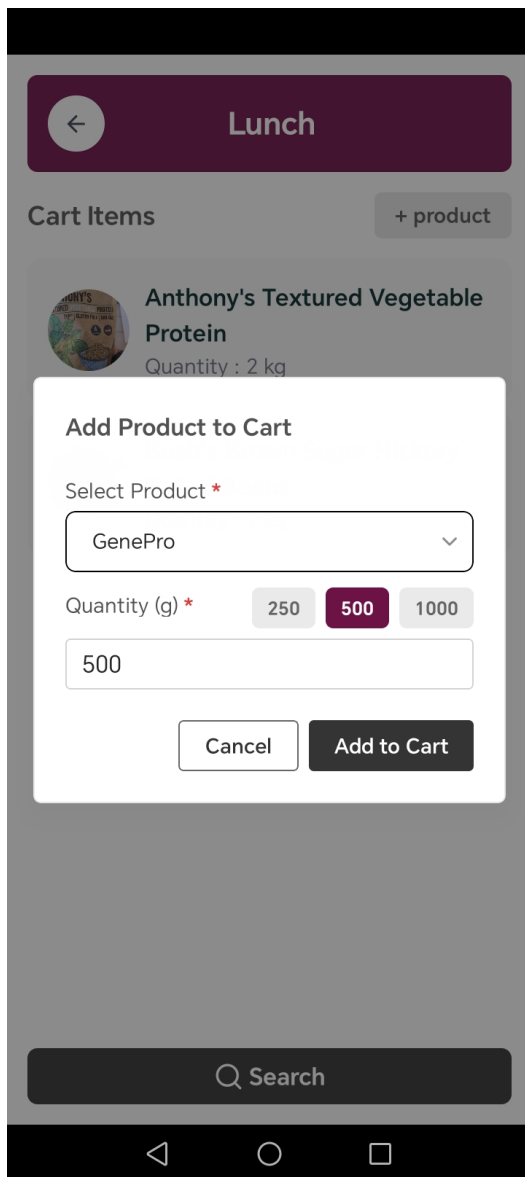


Figure 3.6: User adding a product item to a cart

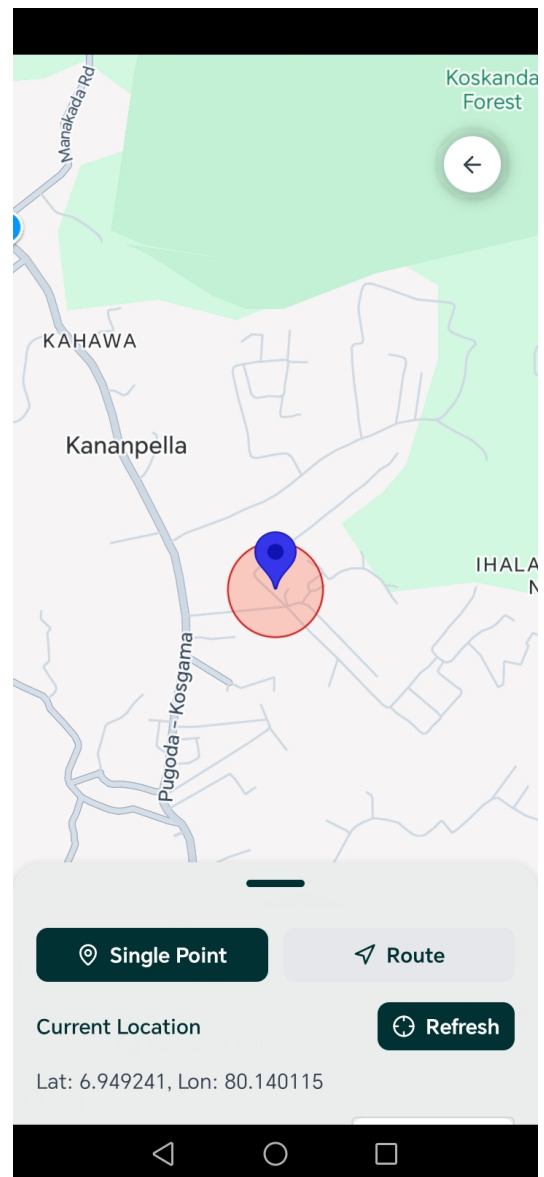


Figure 3.7: User selecting a point on the map

After creating the shopping cart the users can add products and the quantities as shown in the figure 3.6). Then the users can explore store recommendations for the cart. The system will then ask the user to select their location or alternatively select a route for the search as shown by figure (3.7) and figure 3.8) respectively.

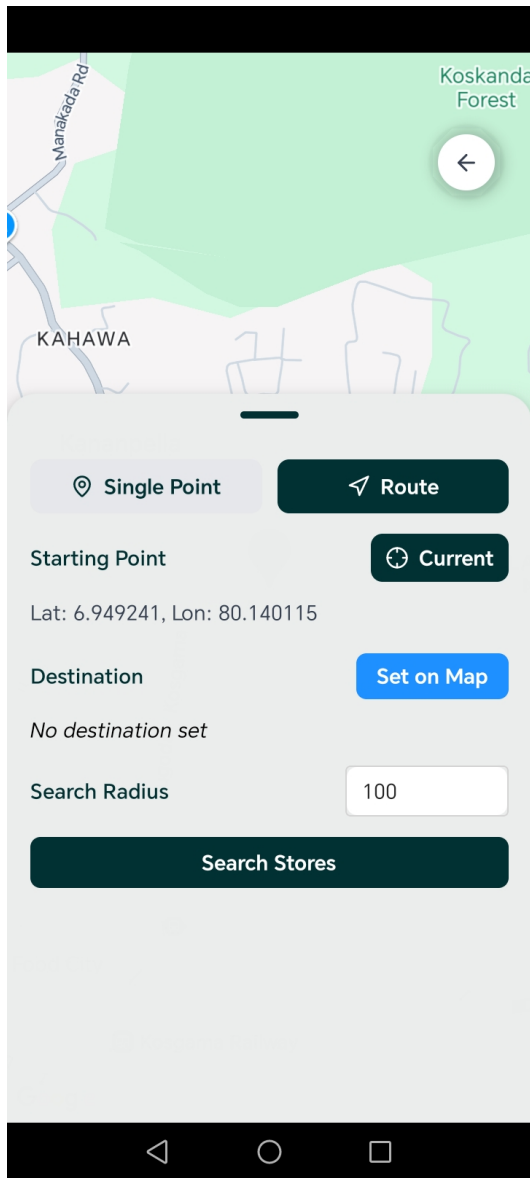


Figure 3.8: User selecting a route on the map to search using a path

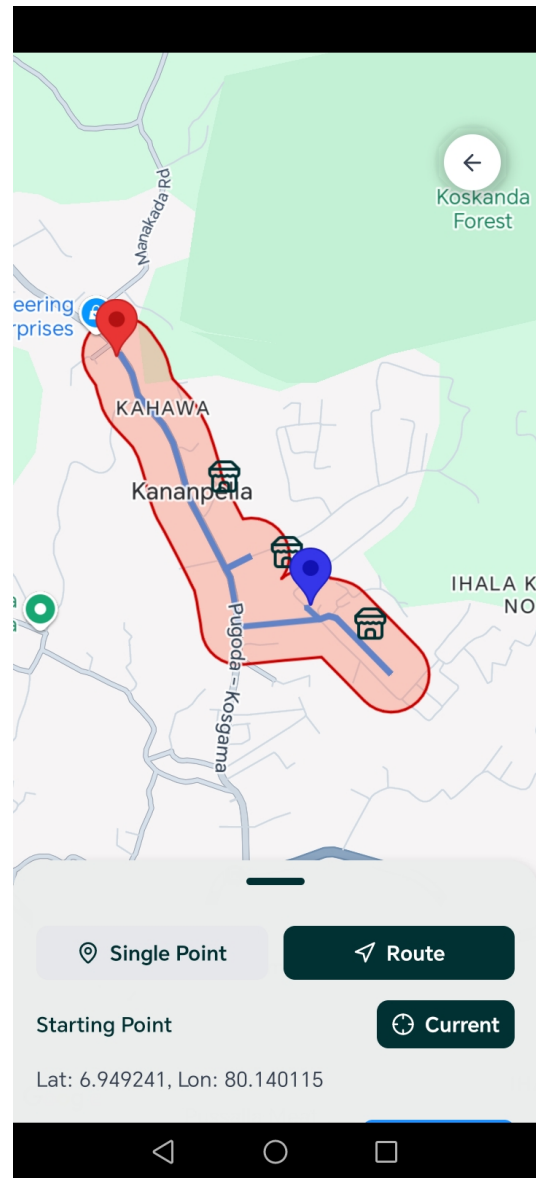


Figure 3.9: User searching for recommendations

Afterwards, the system will give the store recommendations as shown in the figure 3.9). The store icons are the recommendations at which the user can purchase the list of items.

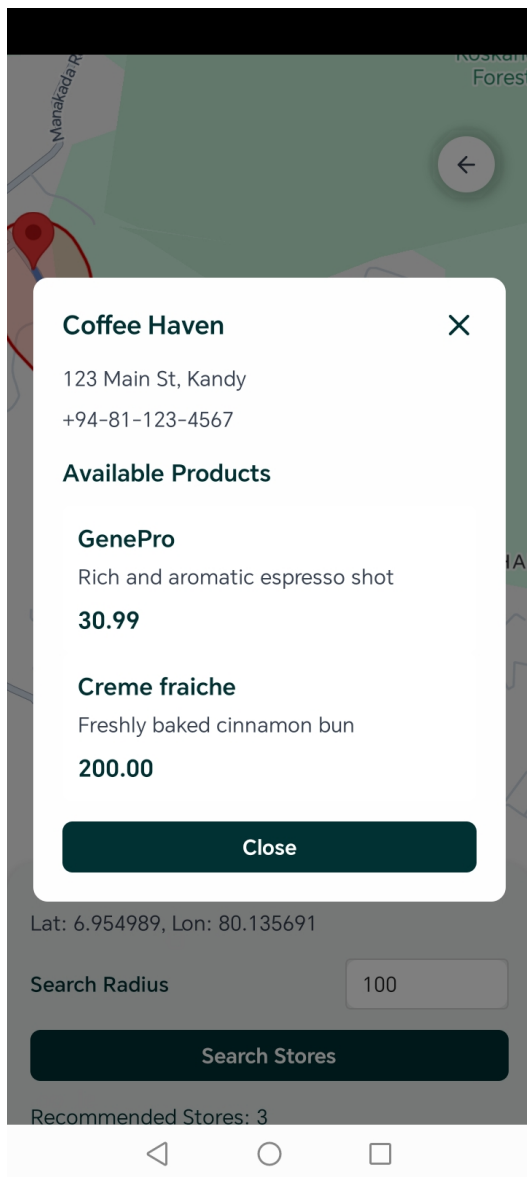


Figure 3.10: User clicking on a store marker

User can then click on the store icons to get the details related to the store as well as the products information related to the items offered by the store as shown in the figure 3.10).

3.2 Research Perspective

Based on the research questions, the recommendation service, route planning and recommendation service, and data collection components for gathering vendor data were identified as research-related components. This section outlines the research efforts, practical considerations, and challenges associated with the methodologies employed to develop solutions for these services.

As stated in the aims and objectives section, the primary goal is to develop a recommendation algorithm, a route planning algorithm, personalized product recommendations, and an efficient data collection method. For the recommendation algorithm, two modes were considered: single-store and multi-store recommendations. The single-store mode involves straightforward filtering and sorting of stores to recommend the top N results, presenting a manageable implementation challenge. Conversely, the multi-store mode, involving subset generation, exhibits exponential time complexity, necessitating research into set cover problem solutions and Traveling Salesman Problem (TSP) optimization algorithms, given the inclusion of distance as a recommendation factor. These are addressed by the solver service.

For data aggregation, the design of REST-based APIs, the potential of web scraping, and the use of IoT or embedded systems were explored. To capture real-world retail dynamics, factors such as store busy hours, queue sizes, proximity to public infrastructure, and discount handling were investigated. Surveys with store owners provided critical insights into practical constraints and requirements, enhancing the research's applicability. Regarding discount handling, due to the complexity of discount conditions, the development of a discount processing engine and a query language for defining discounts was researched. However, as detailed in the implementation section, integrating the discount engine was deferred, as it was deemed a secondary factor rather than a primary driver of recommendations.

To inform the practical design, a survey was conducted to understand operational aspects of grocery stores. The following questions were explored to gather relevant data:

- Operating hours during peak periods (morning, afternoon, evening).
- Store location relative to public infrastructure, such as schools or hospitals.
- Types and applicability of discounts offered.
- Typical queue lengths during busy periods.
- Methods for maintaining sales records, whether digital or manual, and openness to adopting new systems.
- Accepted payment methods.
- Sales trends, restocking frequency, and items requiring frequent restocking.

These survey findings, summarized in the results and analysis section, provided essential context for tailoring the system to real-world retail environments.

Next in line is the personalisation of product suggestions and recommendations, handled by the recommender service. This was done through training a recommender model, and due to the scarcity of data, training data implementation will also be an added task. Research on training and testing recommendation models would have to be conducted as well. Under this section, planned outcomes for this would be to predict the product items that could be (most likely) purchased by the user the next time. The real benefit of this would be to integrate this with the recommendation algorithm to integrate the brand and store preferences as driving factors of recommendations. Using this, the system would be able to find the stores that the user is most likely to visit. The next component is the route planning and managing service. This service's main functionality is to find the most optimal method of reach or the most optimal road system to travel to a store. This algorithm takes the generated set of recommendations from the solver service and tries to give the best possible route to reach the stores. This is also explained in greater detail, including the test data and training data generation, in the coming sections.

In this section, the emphasis on the research requirements was discussed by looking at the system from a very high-level viewpoint, while explaining the main flow of the application through the elaboration of the main use case scenario. In the next section, the discussion will be geared towards going into a much deeper level of each of the services and understanding their methodology and implementation details step by step.

4 Implementation

This section details the internal architecture and implementation of the system’s core components, as introduced earlier. The discussion begins with the store recommendation system, which includes algorithms for both single-store and multi-store recommendation modes, primarily managed by the solver service. Subsequent subsections address the route planning component, user shopping list recommendations, discount engine, and best practices for standardizing APIs. Under the multi-store mode recommendations implementation, multiple algorithms are explored, each presented in its own subsection to accommodate detailed descriptions.

4.1 Store Recommendation System Overview

This subsection introduces the store recommendation problem, providing a formal definition of the challenges associated with each recommendation mode. It further explores the time complexities inherent to these modes and discusses the influence of additional parameters, such as brand and store preferences, on computational complexity.

4.1.1 Problem Definition

The store recommendation system addressed in this dissertation aims to assist users in purchasing a set of desired items from physical stores, optimizing for cost and convenience (mainly traveling distance and the payment method). The system operates in two different modes: *single-store mode*, where all items must be purchased from a single store, and *multi-store mode*, where items can be purchased from multiple stores, but each product can only be bought once and there is only binary selection (either the product is selected from a particular store or not, no partial selections), to minimise total cost, including item prices and travel distance. The multi-store mode is the primary focus due to its combinatorial complexity, which was proven to be NP-hard. This subsection formally defines both modes, demonstrates the NP-hardness of the multi-store problem, discusses general algorithmic approaches for solving such problems in a reasonable time, and examines the impact of additional influence parameters, such as brand or store preferences.

4.1.2 Single-Store Mode

In single-store mode, the user specifies a list of items with required quantities, a location, search radius, a preferred payment method (if the user doesn't mention this, it's taken from the user preferences, otherwise left as non-configured), and the number of desired recommendations, N . The system identifies stores within the radius that have sufficient stock for all items and accept the user's payment method. Each store is evaluated based on the total item cost and the travel distance from the user's location, producing a score (computed using the total cost and a weighted distance). The top N stores with the lowest scores are returned as recommendations.

Formally, let $I = \{i_1, i_2, \dots, i_m\}$ be the set of items, with quantities q_i for each $i \in I$. Let $S = \{s_1, s_2, \dots, s_n\}$ be the set of stores, where each store s_j has:

- Location $l_j = (x_j, y_j)$.
- Stock $k_j : I \rightarrow \mathbb{Z}_{\geq 0}$, where $k_j(i)$ is the available quantity of item i .
- Prices $p_j : I \rightarrow \mathbb{R}_{\geq 0}$, where $p_j(i)$ is the price of item i .
- Payment methods $M_j \subseteq \{\text{credit, cash, } \dots\}$.

The user provides input $U = (I, q, l_u, r, m_u, N)$, where l_u is the user's location, r is the radius, m_u is the payment method, and N is the number of recommendations. A store s_j is *feasible* if:

- $d(l_u, l_j) \leq r$, where d is the Euclidean or road distance.
- $k_j(i) \geq q_i$ for all $i \in I$.
- $m_u \in M_j$.

The score for a feasible store s_j is:

$$\text{Score}(s_j) = \sum_{i \in I} p_j(i) \cdot q_i + w \cdot d(l_u, l_j), \quad (1)$$

where $w \geq 0$ is a weight (e.g., $w = 1$). The goal is to return the top N feasible stores with the lowest scores.

This problem is computationally straightforward, as it involves filtering stores based on the condition that a store must provide all the product in order for that store to be

considered as a possible recommendation, then afterwards the list of recommendations are sorted by score to find the top Y number of recommendations requested by the user, with a complexity of $O(n \cdot m)$ for checking stock and $O(n \log n)$ for sorting, where n is the number of stores and m is the number of items.

4.1.3 Multi-Store Mode

In multi-store mode, the system selects a subset of stores and assigns each item to one store, minimizing the total score, which combines item costs and travel distance across multiple stores. This allows for lower costs by sourcing items from different stores but introduces significant combinatorial complexity. The algorithm can give recommendations with a single store. This is due to the fact that a singleton set is also a subset of all possible subsets of the stores.

Formally, given the same input $U = (I, q, l_u, r, m_u, N)$ and store set S , the system must:

- Select a subset $S' \subseteq S$ such that $d(l_u, l_j) \leq r$ and $m_u \in M_j$ for all $s_j \in S'$.
- Assign each item $i \in I$ to a store $s_j \in S'$ such that $k_j(i) \geq q_i$.
- Compute the total cost $\sum_{i \in I} p_j(i) \cdot q_i$, where s_j is the assigned store for item i .
- Compute the tour distance $d_{\text{tour}}(S')$, defined as the shortest path starting at l_u , visiting each store in S' , and returning to l_u .
- Minimize the score:

$$\text{Score}(S', A) = \sum_{i \in I} p_j(i) \cdot q_i + w \cdot d_{\text{tour}}(S'), \quad (2)$$

where $A : I \rightarrow S'$ is the assignment function mapping items to stores in S' .

The output is the top N solutions (S', A) with the lowest scores, where each solution covers all items and satisfies constraints.

4.1.4 NP-Hardness of Multi-Store Mode

To demonstrate that the multi-store recommendation problem is NP-hard, this algorithm was reduced to two known NP-hard problems: the *Set Cover Problem* and the

Traveling Salesman Problem (TSP).

Reduction from Set Cover: The Set Cover Problem involves selecting the smallest subset (this is important as the fewer number of stores travelled, shorter the traveling distance for each recommendation) of sets from a collection to cover all elements in a universe (In this case, the list of items). Map the multi-store problem to Set Cover as follows:

- Let the items I be the universe of elements.
- Let each store s_j define a set $T_j = \{i \in I \mid k_j(i) \geq q_i, m_u \in M_j\}$, containing items it can supply.
- Set the cost of selecting store s_j to the sum of item prices $\sum_{i \in T_j} p_j(i) \cdot q_i$, and assume $w = 0$ (no travel cost).

A solution to the multi-store problem (selecting stores S' and assigning items) corresponds to selecting a subset of $\{T_j\}$ that covers I with minimum cost. Since Set Cover is a known NP-hard problem, and this can be classified as a special case (no travel distance), hence the multi-store problem is NP-hard.

Reduction from TSP: If all items are available at all stores with equal prices and quantities, the problem reduces to selecting a subset S' that minimizes $d_{\text{tour(traveling distance)}}(S')$. This is equivalent to the TSP (traveling sales men problem), where the goal is to find the shortest tour visiting a subset of nodes (stores) starting and ending at l_u . TSP is also a known NP-hard problem, and due to this reduction of the store allocator problem to the TSP problem, this shows that the multi-store problem's complexity without item assignment constraints.

Since the multi-store problem combines Set Cover (for item coverage) and TSP (for tour distance), it is NP-hard. Verifying a solution (checking stock, payment methods, and computing the tour) can be done in polynomial time, placing the problem in the NP category. Thus, the multi-store recommendation problem can be classified as an NP-complete problem.

4.1.5 General Algorithmic Approaches

NP-hard problems like the multi-store recommendation problem are typically addressed using:

- **Exact Algorithms:** Integer Linear Programming (ILP) or Branch and Bound guarantee optimal solutions but are computationally expensive, suitable for small instances (e.g., $n \leq 20$). In order to find an exact solution, all possible combinations should be evaluated.
- **Heuristic Algorithms:** Greedy algorithms, Genetic algorithms, or Beam Search provide fast, near-optimal solutions for larger instances, trading optimality for scalability. These algorithms will make certain compromises between performance and exact accuracy of the results.
- **Approximation Algorithms:** Algorithms with guaranteed performance ratios (e.g., based on Set Cover or TSP approximations) balance speed and solution quality.

In practice, heuristic and approximation algorithms are preferred for real-time applications, as exact methods become infeasible for large n (e.g., 150 stores), as can be seen in our 4-second timeout constraint.

4.1.6 Impact of Influence Parameters

Adding influence parameters, such as brand preferences or store preferences, significantly increases complexity. Simply put, more conditions and more branches will increase the search space of the problem. For example:

- **Brand Preferences:** If a user prefers specific brands for items, the system must filter stores by brand availability, reducing the feasible store set per item. This adds constraints to the optimisation, increasing the search space for exact algorithms and requiring additional checks in heuristics.
- **Store Preferences:** If users prioritise certain stores (e.g., based on loyalty or ratings), the objective function may include preference weights, transforming the problem into a multi-objective optimisation. This increases runtime by requiring re-evaluation of assignments and may necessitate larger populations in Genetic algorithms or wider beams in Beam Search.

Formally, let $B_i \subseteq \{\text{brand}_1, \text{brand}_2, \dots\}$ be the preferred brands for item i . The constraint $k_j(i) \geq q_i$ becomes $k_j(i) \geq q_i \wedge \text{brand}(i, s_j) \in B_i$, reducing feasible assignments.

For store preferences, introduce weights w_j for each store s_j , modifying the score to:

$$\text{Score}(S', A) = \sum_{i \in I} p_j(i) \cdot q_i \cdot w_j + w \cdot d_{\text{tour}}(S'). \quad (3)$$

This increases the state space in dynamic programming or Branch and Bound (e.g., from $O(2^n)$ to $O(2^n \cdot k)$ for k preference levels) and memory for heuristic populations, impacting scalability.

Next subsections are dedicated to highlighting the implementation details of each and every algorithm explored to overcome the performance barriers explain in the previous sections.

4.2 Single-Store Mode Implementation

The single-store mode is implemented as a pipeline, as shown in Figure 4.1. The process is:

1. **User Query:** Receive $U = (I, q, l_u, r, m_u, N)$.
2. **Filter Nearby Stores:** Identify stores $S' \subseteq S$ where $d(l_u, l_j) \leq r$.
3. **Stock and Payment Check:** Filter S' to stores with $k_j(i) \geq q_i$ for all $i \in I$ and $m_u \in M_j$.
4. **Initial Ranking:** Sort stores by Euclidean distance $d(l_u, l_j)$.
5. **Pruning and Scoring:** Use an OSRM backend to compute actual road distances, caching results in Redis to avoid recomputation. Calculate the score for each store as $\sum_{i \in I} p_j(i) \cdot q_i + w \cdot d_{\text{road}}(l_u, l_j)$.
6. **Output:** Return the top N stores by score.

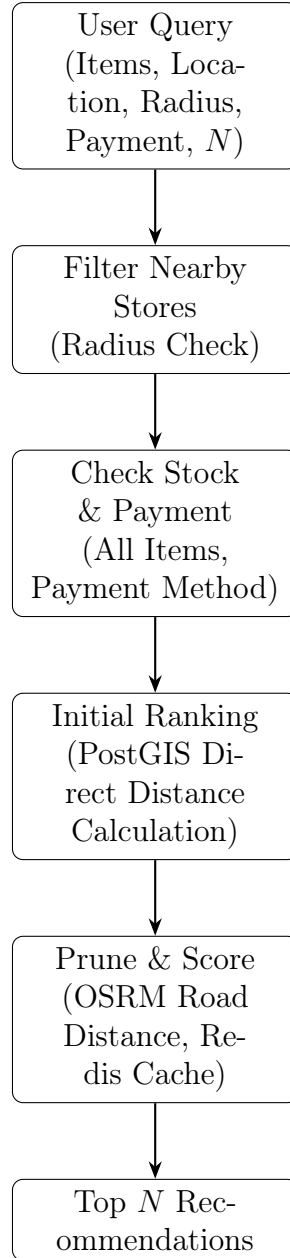


Figure 4.1: Vertical waterfall diagram of the algorithm for single-store mode recommendation.

This pipeline is efficient, with a complexity of $O(n \cdot m + n \log n)$, enhanced by caching for real-time performance. Single-store mode is very straight forward and reflects the use of classic pipe and filter design pattern where the data is filtered in each stage to find only the stores with the product list given by the user. And at the last stage of this pipe line ranking and sorting takes place to find the most optimal solutions. Finally the user will have the top N recommendations required by the user, sorted in the descending order based on the total cost (distance and the cost of product items).

4.3 Multi-Store Mode Implementation

From this point until Subsection 4.10, each subsection is dedicated to exploring a distinct algorithm developed to address the challenges of multi-store recommendations. Each algorithm is accompanied by a detailed description of its design, an analysis of its advantages and disadvantages, a discussion of its limitations, and an overview of the optimizations applied to enhance its performance.

4.4 Naive Algorithm

The naive algorithm was implemented as a straightforward approach that processes a shopping list and categorizes stores based on what items they have in their inventory. It creates a mapping of stores to the items they contain from the shopping list. Then, it categorizes stores into three groups: those with all items, some items, or none of the items on the list. Those categorized stores are then used to find a combination of stores that cover all items in the shopping list. It can either be a single store with all items or a combination of stores with some items. This was done without any optimizations to focus on the correctness of the algorithm. Then that algorithm was used to identify which parts of the algorithm required the most optimization via testing it with several sets of generated data.

A detailed pseudocode for the naive algorithm is provided in Appendix A.1.

4.4.1 Time Complexity Analysis

The `generate_store_recommendation_naive` function iterates through all m inventory items, and for each item, it potentially checks against the n items in the shopping list. This leads to a time complexity of $O(m \cdot n)$ in the worst case. However, if the shopping list items are stored in a hash set for faster look ups, this check becomes $O(1)$ on average, reducing the complexity to $O(m + s \cdot n)$, where s is the number of stores containing at least one item from the shopping list. Subsequently, the categorization of these s stores involves operations proportional to the number of shopping list items, contributing $O(s \cdot n)$ to the overall time.

The `check_combinations` function explores combinations of stores (up to 10) from the set of stores that have some of the required items (s'). The number of such

combinations can grow exponentially with s' , specifically involving a summation of binomial coefficients. For each combination, it checks if all n shopping list items are covered, which takes time proportional to the number of items and the number of stores in the combination. Thus, the time complexity of this function is dominated by the combinatorial search and is approximately $O(\sum_{k=2}^{\min(10, s')} \binom{s'}{k} \cdot k \cdot n)$. The overall time complexity of the naive algorithm is therefore heavily influenced by the combinatorial nature of the store combination checking process.

4.4.2 Space Complexity Analysis

The `generate_store_recommendation_naive` function utilizes space to store the requested item IDs ($O(n)$), a map of stores to their relevant inventory ($O(m)$ in the worst case), and lists of stores categorized by their overlap with the shopping list ($O(s \cdot n)$ in the worst case). Consequently, the space complexity of this function is $O(m + s \cdot n)$.

The `check_combinations` function requires space to store the viable combinations of stores found. The size of this storage depends on the number of successful combinations identified before the early stopping condition is met. Additionally, for each combination being evaluated, a set of remaining items is maintained, requiring $O(n)$ space. Therefore, the space complexity of `check_combinations` is at least $O(n)$ and can increase depending on the number of viable store combinations that need to be stored. The overall space complexity of the algorithm is primarily determined by the storage requirements of the inventory data and the categorized stores in the first function, along with the space needed to store the resulting combinations in the second.

As this has a worst case complexity of $O(n!)$ and as was observed in testing with thousands of generated stores with items it is not feasible to generate satisfactory recommendations using this algorithm. Smaller combinations are cheaper to check and would result in a simpler route so it makes sense to check them first. There are $\sum_{i=0}^k \binom{n}{i}$ possible combinations of k stores. checking each of these combinations is not feasible. That is why the algorithm was adjusted to stop after it finds a given number of valid combinations. Even with that limitation, the algorithm can still take a long time to generate recommendations.

4.4.3 Pros and Cons

Pros:

- **Simple implementation:** The algorithm was straightforward to code and served as a quick way to establish the basic logic of matching shopping lists to store inventories.
- **Correctness verification:** It allowed an initial confirmation of the fundamental correctness of identifying stores with relevant items and exploring their combinations.
- **Bottleneck identification:** Performance testing clearly highlighted the `check_combinations` function as a major computational bottleneck due to its combinatorial nature.
- **Baseline establishment:** The execution time provided a concrete baseline for evaluating the performance improvements of more advanced algorithms.

Cons:

- **Computational inefficiency:** The combinatorial explosion in checking store combinations led to unacceptably long run-times, especially with a larger number of partially matching stores.
- **Scalability issues:** The algorithm's performance degrades rapidly with increasing numbers of stores and potential store combinations, making it unsuitable for large-scale applications.
- **Lack of optimization:** The naive approach lacks any mechanisms to intelligently prune the search space or prioritize more promising store combinations.
- **Redundant computations:** Many redundant combinations were explored, leading to wasted computational resources.
- **Limited practical use:** Poor scalability and efficiency make this naive algorithm impractical for real-world recommendation systems that require timely responses.

Due to the disadvantages observed in the algorithm, it was not selected to be further iterated on in the research. The greatest challenge identified by this algorithm was the combinatorial explosion in checking store combinations. Which led to the development of the **Memory-Optimized Algorithm** and others that will follow.

4.5 Memory-Optimized Algorithm

This approach is reminiscent of the **BLAST Algorithm**, which is used for database searching. The BLAST algorithm is a widely used technique for comparing sequences, which has been found to be particularly efficient for searching large databases. The BLAST algorithm works by comparing a query sequence to a database sequence and identifying regions of similarity between the two sequences. It is then possible to identify the location of the similar regions in the query. Although the BLAST algorithm is not specifically designed for recommendation systems, its approach of comparing sequences and creating similar strings was seen as something that could be adapted to this problem. Thus, this algorithm followed a similar methodology. A descriptive pseudocode for the algorithm is provided in Appendix A.2.

As seen in the code, The items in the stores are stored in a hash set, which is of which the keys are the item IDs encoded to a string and the values are the item quantities. The algorithm then iterates through all the stores, and for each store, checks whether it has the items present in the shopping list. If it does, it adds the shop and the item quantities to one of the result lists flagged as `hasAll`, `hasSome`, or `hasNone`. It then delegates the task of checking the combinations to the `greedy_check_combinations` function. This function goes through ten iterations first to create a power map, which is a mapping of all possible substrings of the item strings to the stores that have them. It then iterates from the shortest substrings to the longest, and for each substring, it finds which items are present and missing from the shopping list. It then checks if a key in the power map exists that matches the missing items. If it does, it adds the combination of stores that have the items to the result list. This process is repeated until the result list is complete. An early return is used in the checking iterations to skip checking combinations that are too long. The result lists are then returned to the main function, which then combines them to generate the final recommendations.

4.5.1 Time Complexity Analysis

The `to_string(array)` function operates in linear time, $O(l)$, where l represents the length of the input array. The `pre_calculate_combinations()` function has a time complexity of $O(m + S \cdot (I_{max} \log I_{max}))$, where m is the total number of items, S is the number of stores, and I_{max} is the maximum number of unique items in any single store; crucially, due to caching, this function executes only once. The `generate_store_recommendation_memory()` function's time complexity is $O(n \log n + S \cdot n \cdot L_s + \text{complexity of } greedy_check_combinations)$, where n is the number of items in the shopping list, and L_s is the average length of the item set string; the `greedy_check_combinations()` function's complexity is considered separately. The `non_empty_substrings(string)` function has a time complexity of $O(l^3)$, where l is the length of the input string. The `greedy_check_combinations()` function has a time complexity that is potentially exponential in the number of shopping list items due to the generation of substrings and the exploration of the power set of item combinations.

4.5.2 Space Complexity Analysis

The `to_string(array)` function uses $O(l)$ space. The `pre_calculate_combinations()` function uses $O(S \cdot I_{max})$ space to store the mapping of item set strings to store IDs. The `generate_store_recommendation_memory()` function uses, in the worst case, $O(2^n \cdot S)$ space for `stores_with_matching_items`. The `non_empty_substrings(string)` function uses $O(l^3)$ space. The `greedy_check_combinations()` function uses $O(n^2 \cdot S)$ space for `store_item_power_map`.

4.5.3 Pros and Cons

Pros:

- **Pre-calculation:** Caching store-item combinations in `pre_calculate_combinations` can save computation for repeated recommendations with static inventory.
- **String-based comparison:** Using string representations might offer faster item set comparisons.

- **Greedy combination check:** `greedy_check_combinations` attempts to find covering combinations without exhaustive search.
- **Subset focus:** Exploring item subsets in `greedy_check_combinations` could be more efficient than individual item combinations.

Cons:

- **Limitation of string representation:** The string representation of item IDs might not be suitable for all use cases.
- **High time complexity:** `greedy_check_combinations` has a potentially exponential time complexity in the number of items in the shopping list.
- **High space complexity:** Storage for intermediate data structures like `store_item_power_map` and `stores_with_matching_items` can be significant.
- **Greedy optimality not guaranteed:** The greedy approach in combination checking might not find the most optimal solutions (for example, fewest stores).
- **String conversion overhead:** Repeated string conversion and sorting could introduce overhead.
- **Complexity of greedy check:** The logic within `greedy_check_combinations` is complex and contributes to the overall algorithm's intricacy.

The text comparison approach showed some promise in the initial developments as it calculated combinations much faster than the naive approach. However, the text comparison approach was not suitable for the problem as the results were not so significant compared to the other approaches that were tried. The algorithm would end up taking a long time to generate recommendations similar to the naive approach as the number of stores and items increased. As can be seen in the end of the pseudo code, to get optimal results, the store item power set would need to be expanded with pairs as the keys, then with triplets as the keys and so on. This would lead to an exponential increase in space complexity and the implementation would be difficult in reality to be performant. This in addition to the fact that it runs into a hard limit

at the character limit to the items in stores made the text comparison approach not suitable for the problem. However the increase in efficiency given by the pre-calculation was noted to be beneficial to the system in the long run. Instead of focusing on the combinatorial problem, the next approaches were more focused on getting the most optimal results via selecting the most promising store combinations and dropping any irrelevant ones.

4.6 Filtered Greedy Algorithm

This algorithm used the greedy approach used in the previous algorithms much more efficiently by filtering out everything unrelated to the shopping list. It showed promise in the initial stages and was developed further to improve the efficiency of the algorithm. This is the algorithm that was first implemented with a database connection to the inventory. The previous algorithms were implemented with a static inventory that was generated in the code. This algorithm was implemented with a database connection to a generated inventory. Even with the database connection, the algorithm was much more efficient than the previous algorithms. So additional criteria like item quantities could be added to it without compromising the efficiency. A detailed pseudocode for the Filtered Greedy Algorithm is provided in Appendix A.3.

4.6.1 Time Complexity Analysis

Initially, filtering the store inventory based on the shopping list items requires time proportional to the total number of items across all stores, denoted as M . For each of the n items in the shopping list, checking its presence involves an average time complexity of $O(1)$ with efficient lookups, leading to a filtering step of roughly $O(M)$. Subsequently, adjusting the quantities and prices for each of the n shopping list items might involve iterating through a subset of the store inventory, potentially reaching $O(n \cdot M)$ in a naive scenario. However, efficient indexing of the `store_inventory` could improve this. Grouping the items by store ID and calculating the total prices and item sets involves processing the filtered inventory, which contains at most M entries. This step can take approximately $O(M \log M)$ if sorting is used for grouping, or $O(M)$ on average with hash maps, followed by $O(M)$ for the calculations. Categorizing the stores based on whether they have all or some of the requested items takes time proportional

to the number of stores, S , resulting in $O(S)$. Sorting the `store_summaries` by item count requires $O(S \log S)$, and sorting the `stores_with_all_items` by price takes $O(S' \log S')$, where S' is the number of stores with all items. The most complex part is finding combinations of stores, which involves nested loops. The outer loop iterates up to $n - 2$ times, and the inner loop iterates through the stores with some items, potentially $O(S)$ in the worst case. Inside these loops, finding missing items and then searching for stores that carry these missing items in the `store_inventory` can lead to a time complexity approaching $O(n^2 \cdot S \cdot M)$ in a loose upper bound without considering data structure optimizations. However, the early exit condition with the `break inner` loop makes the actual performance better in practice. Overall, the nested loops for finding store combinations are dominant factor in the worst-case time complexity.

4.6.2 Space Complexity Analysis

The space complexity of the algorithm depends on the storage requirements of several data structures. The `requested_item_ids` list requires $O(n)$ space. The `store_inventory` in the worst case can hold up to M items that are present in the shopping list. The `store_summaries` structure stores information for each store having at least one item from the shopping list, potentially requiring $O(S \cdot n)$ in the worst case. Similarly, `stores_with_all_items` and `stores_with_some_items` can collectively use up to $O(S \cdot n)$ space. The `requested_items_set` needs $O(n)$ space. The `recommendations` structure's size is contingent on the number of combinations found, and each combination might store information about multiple stores and up to n items, with the `additional_stores` dictionary potentially holding up to S entries. Consequently, the space required for `recommendations` can be significant. Therefore, the dominant space complexity is determined by the size of the filtered `store_inventory` ($O(M)$) and the potential size of the `recommendations` structure, leading to a rough upper bound of $O(M + S \cdot n)$.

4.6.3 Pros and Cons

Pros:

- **Filtering:** The algorithm efficiently filters the inventory to only consider items present in the shopping list, significantly reducing the data to process.

- **Quantity and price adjustment:** Incorporating quantity requirements and calculating the price based on these requirements early in the process adds practical value to the recommendations.
- **Greedy approach with categorization:** The algorithm categorizes stores into those with all and some items, and then greedily tries to find combinations to fulfill the remaining items, which can be more efficient than exploring all combinations.
- **Prioritization by item count and price:** Sorting stores by the number of matching items and then by price for stores with all items helps in suggesting more relevant and potentially cheaper options first.
- **Database integration:** The algorithm is designed to work with a database, making it more practical for real-world scenarios with large inventories.

Cons:

- **Greedy optimality is not guaranteed:** The greedy approach to finding combinations of stores might not always yield the combination with the fewest number of stores or the absolute lowest total price.
- **Potential inefficiency in combination search:** While more efficient than the naive approach, the nested loops for finding store combinations for missing items could still be computationally intensive in scenarios with many stores having subsets of the required items. The early **break** helps, but the worst-case complexity could still be high.
- **Complexity of logic:** The logic for finding combinations and tracking additional stores can be complex to implement and debug.
- **Limited exploration of combinations:** The algorithm seems to stop after finding the first set of additional stores for a given main store and item count, which might prevent it from finding other potentially better combinations.
- **Performance dependence on data distribution:** The performance of the combination finding step can heavily depend on how the required items are distributed across different stores.

As mentioned earlier, the **Filtered Greedy Algorithm** was the first approach that was implemented with a database connection to the inventory. This algorithm also faces a few challenges even without the database connection as can be seen in the pseudo code. The implementation of the combinatorial problem in the algorithm was not fit for finding the most optimal results. The algorithm, when checking for stores to fill the missing items, would only add the store which has the item at the best price, this does not take into account the other stores that may have more than one item at potentially, a lower combined price. This is an inherent problem with the greedy approach and the algorithm would not be able to find the most optimal results. Expanding the algorithm to include the other stores that have the same item would lead to a combinatorial explosion. The algorithm as is was already a not very well performing one and would take a long time to generate recommendations (less than the previous algorithms but still slow). The database connection is a necessity in the project as a static inventory will never exist in reality. Due to these challenges, this algorithm was not selected to be further iterated on. Rather, the focus went to more aggressive pruning of the search space and the use of better algorithms to find the most optimal results. Until this point, the algorithms had not considered the fact that more factors like distance between stores, the resulting route distance, and the user's preference to stick to as few stores as possible, would need to be factored into the recommendation process. Algorithms following this section were tried out in a more practical setting using real-world store data and generated store inventory data.

The multi-store mode, which involves selecting a subset of stores and assigning items to minimize total cost and travel distance, is NP-hard due to its combination of Set Cover and Traveling Salesman Problem (TSP) complexities. When looking at the survey results, some common recurring patterns can be found in the item purchases of customers. For example, the number of stores a customer is willing to visit at a time and the number of items purchased, along with other metrics, provide a way to reduce the size of the search space. In this section, some algorithms that are discussed take advantage of these factors to provide more practical answers. Hence, most of the algorithms guarantee optimal solutions for small to medium instances.

Under each of these algorithms, some keywords are used extensively, such as 4-seconds, pipeline and similarities between many algorithms and their processes (pipelines) can

be found. These were intentionally left for the reader to properly grasp the evaluation processes and the time constraints or boundaries the system is trying to achieve to meet the real-time constraints to reduce the latency.

Each algorithm follows a similar structure, they all require database connection (PostgreSQL server) (this is only valid for the real-world related implementation details, base algorithms explained in this section are concerned with explaining the details of the algorithms themselves rather than delving into the pipe-and-filter structure of each of the algorithms), connection to an in-memory cache (Redis server), and Open Street Routing Machine(OSRM) server to calculate traveling distances in real-world road systems(more specifically road systems in Sri Lanka).

4.7 Branch and Bound Algorithm

The Branch and Bound algorithm is an exact method, based on the branch and bound technique (*Branch and bound* 2025) designed to solve the multi-store recommendation problem by exploring the solution space while pruning suboptimal branches. The goal is to find the top N subsets of stores $S' \subseteq S$ and item assignments $A : I \rightarrow S'$ that minimize the score:

$$\text{Score}(S', A) = \sum_{i \in I} p_j(i) \cdot q_i + w \cdot d_{\text{tour}}(S'), \quad (4)$$

where $p_j(i)$ is the price of item i at store $s_j \in S'$, q_i is the required quantity, $w \geq 0$ is a distance weight (set to 1 in this work), and $d_{\text{tour}}(S')$ is the shortest tour distance starting and ending at the user's location l_u , visiting each store in S' . The algorithm ensures all constraints are satisfied: stores must be within the user's radius r , have sufficient stock ($k_j(i) \geq q_i$), and accept the payment method m_u .

The algorithm operates by constructing a search tree where each node represents a partial solution, defined by a subset of selected stores S' , a set of covered items (via a bitset), and an item-to-store assignment. The root node starts with an empty store set and no items covered. At each level, the algorithm branches by adding a new store to the current subset, updating the covered items and assignment. To reduce the exponential search space ($O(2^n)$ for n stores), it employs:

- **Lower Bound Pruning:** For each partial solution, a lower bound on the score

is computed as the current total cost (items assigned so far) plus the current tour distance (approximated via a Minimum Spanning Tree, MST) plus the minimum cost of uncovered items (cheapest price per item across all feasible stores). If this lower bound exceeds the best-known score, the branch is pruned.

- **Bitset for Coverage:** A bitset tracks covered items, enabling efficient union operations when adding stores and checking if all items are covered ($S' \cap I = I$).
- **Duplicate Avoidance:** A set of visited store combinations (as sorted tuples) prevents redundant solutions.

The algorithm terminates when all feasible solutions are explored or a timeout (e.g., 4 seconds) is reached, returning the top N solutions sorted by score. A detailed pseudocode outlining the core logic, focusing on the combinatorial structure, is provided in Appendix A.4.

The algorithm’s complexity is exponential, $O(2^n \cdot m)$ in the worst case, where n is the number of stores and m is the number of items, due to the potential exploration of all store subsets and item assignments. However, pruning significantly reduces the search space for practical instances (e.g., $n \leq 20$). The space complexity is $O(n + m)$ for the recursion stack and bitsets, plus $O(n^2)$ for the distance matrix.

4.7.1 Pipeline

The Branch and Bound algorithm is integrated into a pipeline that processes user input and delivers recommendations in real-time, adhering to the 4-second timeout constraint. The pipeline, tailored to the multi-store mode, consists of the following steps:

1. **User Query:** The system receives the user input $U = (I, q, l_u, r, m_u, N)$, specifying items, quantities, location, radius, payment method, and number of recommendations.
2. **Store Fetching:** A PostgreSQL database is queried to retrieve stores within radius r of l_u , with sufficient stock for at least one item in I and supporting m_u . The query uses PostGIS for geospatial filtering (e.g., `ST_DWithin`) and aggregates

stock and price data, limiting results to 150 stores to ensure scalability. Unit conversions (e.g., kg to g) are handled to compute standardized prices.

3. **Distance Precomputation:** Road distances between the user’s location and stores are kept in an in-memory cache (not on Redis); later, for frequent requests, data will be cached in Redis, and also the distance data among stores are fetched from an OSRM (Open Street Routing Machine) backend server through the table API and cached in Redis for efficiency. A distance matrix is constructed to support fast lookups during algorithm execution.
4. **Data Preprocessing:** The algorithm initializes data structures:
 - A bitset for each store, encoding available items.
 - A mapping of items to feasible stores, sorted by price.
 - Minimum item costs for lower bound estimation.
5. **Algorithm Execution:** The Branch and Bound algorithm is executed with a timeout (4 seconds), exploring the search tree and pruning based on the lower bound. The MST-based tour distance approximation (doubling the MST weight) ensures efficient distance calculations.
6. **Post-Processing:** The top N recommendations are sorted by score and formatted as (S', A) pairs, with store IDs, item assignments, total cost, tour distance, and score.
7. **Output:** Recommendations are returned to the user, with metrics (number of stores, number of recommendations, total cost, total distance and score) logged to a CSV file for evaluation.

The pipeline leverages caching (Redis) and efficient database queries to minimize latency, ensuring the algorithm’s computational overhead is the primary bottleneck. The vertical waterfall diagram for the single-store mode (Figure 4.1) can be adapted for the multi-store mode by replacing the ranking and pruning steps with the Branch and Bound execution. Pipeline of the parallelized Branch and Bound algorithm is illustrated in the figure 4.2. In the parallelized version the pipeline enhances Branch

and Bound by distributing search subtrees across CPU cores, with a modified post-processing step to merge worker results.

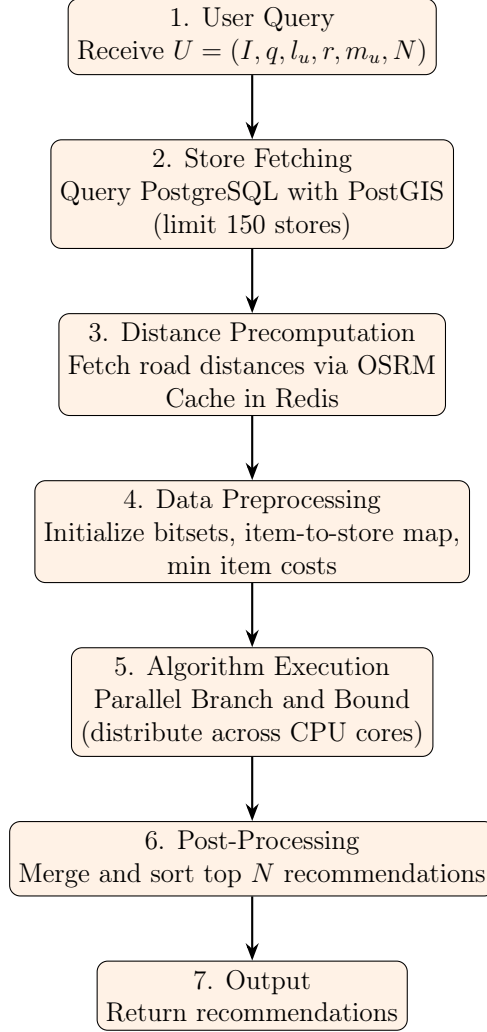


Figure 4.2: Vertical waterfall diagram of the Parallel Branch and Bound algorithm.

4.7.2 Optimization Steps

The base Branch and Bound algorithm, while exact, is computationally expensive for large instances ($n > 20$). Several optimizations, implemented in Go variants, improve its performance:

- **Beam Search Integration:** The optimized version incorporates a beam search heuristic, maintaining only the top $K = 1000$ partial solutions at each level of the search tree, sorted by lower bound. This reduces the search space to $O(K \cdot n \cdot m)$, trading optimality for scalability.

- **Parallelization:** The parallel version distributes the search tree across multiple CPU cores. Initial states (single-store solutions) are partitioned among workers, each exploring an independent subtree. A Redis pub/sub channel synchronizes the global best score, allowing workers to prune branches dynamically. The MST computation is also parallelized using a thread-safe priority queue, utilizing all available cores (e.g., `runtime.NumCPU()`).
- **Efficient Distance Caching:** All versions use a precomputed distance matrix stored in Redis, with OSRM queries batched to minimize network overhead. The parallel version adds validation to ensure complete distance data, improving robustness.
- **Early Feasibility Checks:** The algorithm checks for single-store solutions (where one store covers all items) before branching, returning immediately if found, reducing runtime for simple cases.

These optimizations enable the algorithm to handle larger instances (e.g., $n = 150$) within the 4-second timeout, with the parallel version achieving near-linear speedup with additional cores.

4.7.3 Pros and Cons

Pros:

- **Optimality:** The base algorithm guarantees the optimal solution for small instances, making it ideal for processing smaller and medium-sized instances and benchmarks of other heuristics.
- **Flexibility:** Easily adapts to additional constraints (e.g., brand preferences) by modifying the lower bound or branching logic.
- **Robust Pruning:** The MST-based lower bound and bitset operations efficiently prune the search space, reducing runtime for practical cases.
- **Scalable Optimizations:** Beam search and parallelization extend the algorithm’s applicability to larger datasets, balancing speed and quality. In the parallel version, computational tasks can be further delegated to one or more

machines or a cluster of machines based on the size of the search space. Due to this, the branch and bound algorithm becomes one of the most scalable algorithms.

Cons:

- **Exponential Complexity:** The base algorithm's $O(2^n \cdot m)$ complexity limits its use to small instances without optimizations.
- **Memory Overhead:** The recursion stack and candidate storage can consume significant memory for large n , especially before pruning.
- **Timeout Sensitivity:** The 4-second timeout may truncate exploration, returning suboptimal or no solutions for complex inputs.
- **Distance Approximation:** The MST-based tour distance doubles the MST weight, potentially overestimating the true TSP distance, affecting score accuracy. It requires extensive analysis of travel data to find an optimal constant for recommendation scenarios.
- **Aggressive Pruning:** Even though pruning is preferred to reduce the search space, it could lead to the removal of valid recommendations. Thus, operational constraints of these algorithms are much tighter and deviate from theoretical constraints and shift towards practical scenarios.

4.7.4 Limitations and Future Improvements

The Branch and Bound algorithm faces several limitations, particularly in scalability and real-time performance:

- **Scalability:** For $n > 20$, the exponential growth of the search tree overwhelms even optimized versions, especially with many items (m). The beam search variant sacrifices optimality, and parallelization is limited by CPU cores.
- **Timeout Constraints:** The 4-second timeout, critical for real-time applications, often interrupts exploration, especially for inputs with many stores or items.

- **Distance Estimation:** The MST approximation, while efficient, may lead to suboptimal pruning if the true TSP distance is significantly lower.
- **Synchronization Overhead:** In the parallel version, Redis pub/sub and mutex locks introduce minor overhead, reducing speedup for small instances.

Future improvements could address these issues:

- **Distributed Computing:** Extend parallelization to a distributed system (e.g., using a message queue like RabbitMQ) to handle very large datasets ($n = 500$), partitioning the search tree across machines.
- **Hybrid Algorithms:** Combine Branch and Bound with heuristics (e.g., Greedy or Genetic algorithms) to warm-start the search with high-quality initial solutions, reducing exploration time.
- **Improved Distance Bounds:** Replace the MST approximation with a Held-Karp lower bound for TSP, providing tighter estimates and more effective pruning.
- **Dynamic Timeout Adjustment:** Implement adaptive timeouts based on input size (e.g., n, m) or use incremental solution output to ensure at least partial results within 4 seconds.
- **Memory Optimization:** Use iterative deepening or a priority queue for state exploration to reduce recursion stack size, enabling larger instances on limited hardware.

These enhancements would make the algorithm more robust for real-world retail applications, balancing optimality, speed, and scalability. The Branch and Bound algorithm serves as a strong foundation for exact solutions.

4.8 Beam Search Algorithm

The Beam Search algorithm is a heuristic approach designed to address the scalability limitations of the Branch and Bound algorithm (Subsection 4.7) for the multi-store recommendation problem, as defined in Subsection 4.1.1. By maintaining a fixed-size

beam of promising partial solutions, it efficiently explores the solution space, trading optimality for speed. This subsection describes the algorithm theoretically, outlines its pipeline, discusses optimization steps, evaluates its pros and cons, and identifies limitations with future improvements.

4.8.1 Algorithm Description

The Beam Search algorithm tackles the NP-hard multi-store recommendation problem by selecting a subset of stores $S' \subseteq S$ and assigning items $A : I \rightarrow S'$ to minimize the score:

$$\text{Score}(S', A) = \sum_{i \in I} p_j(i) \cdot q_i + w \cdot d_{\text{tour}}(S'), \quad (5)$$

where $p_j(i)$ is the price of item i at store $s_j \in S'$, q_i is the quantity, $w = 1$ is the distance weight, and $d_{\text{tour}}(S')$ is the shortest tour distance starting and ending at the user's location l_u , visiting each store in S' . Constraints include store proximity (within radius r), sufficient stock ($k_j(i) \geq q_i$), and payment method compatibility (m_u).

Unlike Branch and Bound's exhaustive search, Beam Search maintains a beam of at most $K = 100$ partial solutions at each level of the search tree. Each node represents a partial solution: a subset of stores S' , a bitset of covered items, an item-to-store assignment, and the current cost. The algorithm starts with an empty store set and iteratively expands the beam by adding one store per branch, updating covered items and assignments. Key features include:

- **Beam Pruning:** After expanding all nodes in the current beam, only the top K new states (sorted by estimated score = current cost + tour distance) are retained, reducing the search space to $O(K \cdot n \cdot m)$.
- **Lower Bound Estimation:** For each partial solution, a lower bound is computed as the current cost plus the current tour distance (approximated via a Minimum Spanning Tree, MST) plus the minimum cost of uncovered items. Infeasible branches (items with no feasible stores) are pruned.
- **Bitset Operations:** A bitset tracks covered items, enabling efficient union and difference operations.

- **Duplicate Avoidance:** A set of visited store combinations (sorted tuples) prevents redundant solutions.

The algorithm terminates when all items are covered in a solution or a timeout (4 seconds) is reached, returning the top N recommendations sorted by score. A detailed pseudocode outlining the core logic is provided in Appendix A.5.

The time complexity is $O(K \cdot n \cdot m)$, where K is the beam width, n is the number of stores, and m is the number of items, as each level processes up to K states, each branching to n stores and updating m items. The space complexity is $O(K \cdot (n + m))$ for the beam (storing store IDs, bitsets, and assignments) plus $O(n^2)$ for the distance matrix.

4.8.2 Pipeline

The Beam Search algorithm is integrated into a pipeline that processes user input and delivers recommendations in real-time, adhering to the 4-second timeout constraint. The pipeline, tailored to the multi-store mode, is similar to that of Branch and Bound (Subsection 4.7.1) but uses Beam Search for the execution step. The steps are:

1. **User Query:** Receive the user input $U = (I, q, l_u, r, m_u, N)$, specifying items, quantities, location, radius, payment method, and number of recommendations.
2. **Store Fetching:** Query a PostgreSQL database with PostGIS to retrieve stores within radius r of l_u , with sufficient stock for at least one item in I and supporting m_u . Limit to 150 stores for scalability, handling unit conversions for standardized prices.
3. **Distance Precomputation:** Fetch road distances from an OSRM (Open Street Routing Machine) backend server through the table API and cache in Redis. Construct a distance matrix for fast lookups.
4. **Data Preprocessing:** Initialize bitsets for store items, a sorted item-to-store mapping, and minimum item costs for lower bound estimation.
5. **Algorithm Execution:** Execute Beam Search with a beam width of $K = 100$, exploring the search tree and retaining the top K states per level within the 4-second timeout.

6. **Post-Processing:** Sort the top N recommendations by score and format as (S', A) pairs, including store IDs, item assignments, total cost, tour distance, and score.
7. **Output:** Return recommendations to the user, logging runtime and solution metrics for evaluation.

The pipeline is visualized in Figure 4.4, a vertical waterfall diagram illustrating the flow from user input to recommendations. Caching (Redis) and efficient queries ensure low latency, with the Beam Search execution being the primary computational bottleneck. The pipeline processes user input through database queries, distance caching, and heuristic search with a beam width of $K = 100$ to deliver near-optimal recommendations within a 4-second timeout.

4.8.3 Optimization Steps

The base Beam Search algorithm (implemented in Python) is optimized for efficiency but faces scalability challenges for large instances ($n > 100$). The following optimizations, including those from a Go implementation, enhance performance:

- **Efficient Data Structures:** The Python version uses bitsets for item coverage and sorted item-to-store mappings, reducing the cost of checking coverage and selecting the cheapest stores. This minimizes redundant computations during beam expansion.
- **Parallel Beam Exploration:** A Go implementation parallelizes beam expansion by distributing the beam across CPU cores. Each worker processes a subset of the current beam's states, generating new states and synchronizing the global best score via a Redis pub/sub channel. The MST computation is also parallelized using a thread-safe priority queue, leveraging all available cores.
- **Distance Caching:** Both implementations use a precomputed distance matrix stored in Redis, with batched OSRM queries to minimize network overhead. The Go version validates distance data completeness for robustness.

- **Early Feasibility Checks:** The algorithm checks for single-store solutions before expanding the beam, returning immediately if a store covers all items, reducing runtime for simple cases.

These optimizations enable the algorithm to handle larger instances (e.g., $n = 150$) within the 4-second timeout, with the parallel Go version achieving significant speedup on multi-core systems.

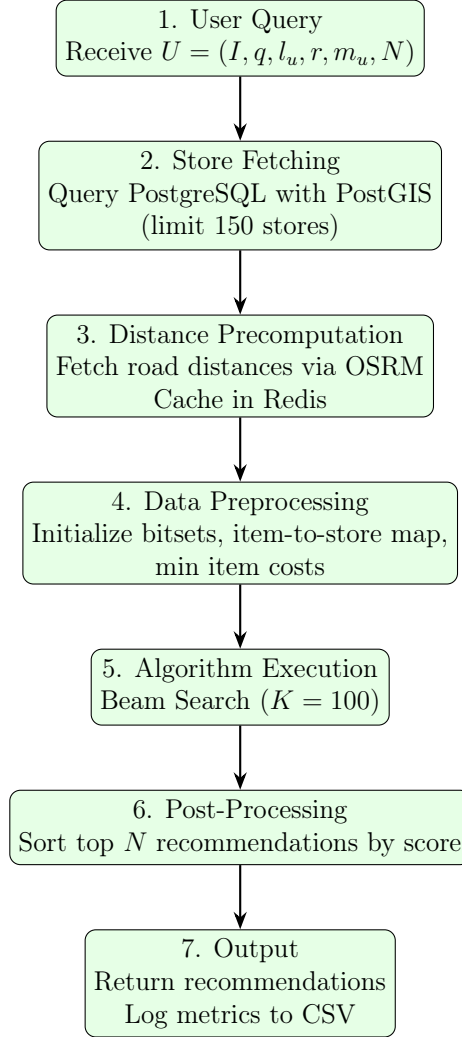


Figure 4.3: Vertical waterfall diagram of the Beam Search algorithm.

4.8.4 Pros and Cons

Pros:

- **Scalability:** The fixed beam width ($K = 100$) reduces the search space to $O(K \cdot n \cdot m)$, enabling faster execution than Branch and Bound for large n .

- **Near-Optimal Solutions:** Beam Search often finds high-quality solutions close to the optimal, suitable for real-time applications.
- **Flexibility:** Easily incorporates additional constraints (e.g., brand preferences) by adjusting the lower bound or beam sorting criteria.
- **Parallelization Potential:** The Go implementation’s parallel exploration leverages multi-core systems, improving runtime for complex inputs.

Cons:

- **Sub-optimality:** The heuristic nature of Beam Search may miss the optimal solution, especially with a small beam width.
- **Beam Width Sensitivity:** The quality of solutions depends heavily on K , with small K leading to poorer solutions and large K increasing memory and runtime.
- **Memory Usage:** Storing K states, each with store IDs, bitsets, and assignments, can be memory-intensive for large K or m .
- **Distance Approximation:** The MST-based tour distance overestimates the true TSP distance, potentially skewing the lower bound and solution ranking.
- **Limits Possibility of Multi-Store Recommendations:** This algorithm is always trying to find and recommend users sets of stores containing a single store. This behavior (Note: not the results) is identical to the number of recommendations given by the branch and bound algorithm optimized using beamer search.

4.8.5 Limitations and Future Improvements

The Beam Search algorithm, while scalable, has limitations that impact its performance in certain scenarios:

- **Suboptimality:** The fixed beam width sacrifices optimality, particularly for complex inputs with many stores or items, where the optimal solution may be pruned early.

- **Timeout Sensitivity:** The 4-second timeout may interrupt exploration, especially for large n or m , returning incomplete or suboptimal solutions.
- **Distance Estimation:** Similar to the branch and bound algorithm, the MST approximation overestimates the TSP distance, leading to suboptimal pruning or ranking of solutions.

Future improvements could enhance the algorithm’s effectiveness:

- **Dynamic Beam Width:** Adjust K dynamically based on input size or runtime, increasing K for small instances to improve solution quality and decreasing it for large instances to ensure completion.
- **Distributed Computing:** Extend parallelization to a distributed system (e.g., using RabbitMQ) to handle very large datasets ($n = 500$), partitioning the beam across machines.
- **Hybrid Algorithms:** Combine Beam Search with local search (e.g., 2-opt for TSP) or Genetic algorithms to refine solutions post-beam exploration, improving score quality.
- **Improved Distance Bounds:** Use a Held-Karp lower bound for TSP to provide tighter distance estimates, enhancing pruning accuracy.
- **Incremental Output:** Implement progressive solution output to return partial results within the 4-second timeout, ensuring usability for time-constrained scenarios.

These enhancements would make Beam Search more robust for real-time retail applications, building on its scalability to address the multi-store problem effectively. This algorithm serves as a practical alternative to exact methods. Further more Beam Search can be used with Branch and Bound technique by changing the pipeline of the Branch and Bound algorithm to incorporate Beam Search for reducing the search space. This is illustrated in the figure 4.4. The pipeline is identical to Branch and Bound, except for the algorithm execution step, which uses a beam width of $K = 1000$ to limit exploration for scalability.

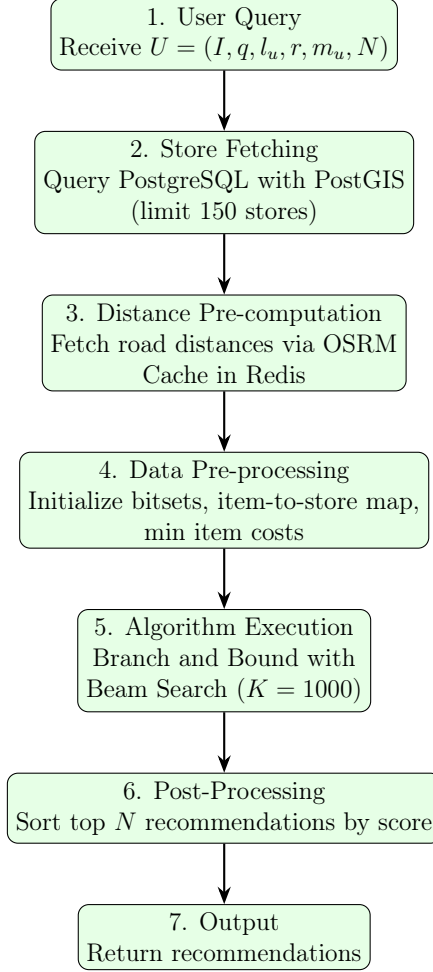


Figure 4.4: Vertical waterfall diagram of the optimized Branch and Bound algorithm.

4.9 Integer Linear Programming

Integer Linear Programming (ILP) provides an exact solution to the multi-store recommendation problem, as defined in Subsection 4.1.1, by formulating it as a constrained optimization problem solved using the Gurobi solver. Unlike the Beam Search (Subsection 4.8), ILP guarantees optimality but at a higher computational cost. Compared to Branch and Bound (Subsection 4.7), ILP uses a mathematical programming approach rather than explicit tree search. Implemented in Python, it leverages PostgreSQL for store data, Redis for distance caching, and OSRM for tour distances. This subsection describes the ILP formulation, outlines its pipeline, discusses optimization steps, evaluates its pros and cons, and identifies limitations with future improvements.

4.9.1 Algorithm Description

The ILP selects a subset of stores $S' \subseteq S$ and assigns items $A : I \rightarrow S'$ to minimize the score:

$$\text{Score}(S', A) = \sum_{i \in I} p_j(i) \cdot q_i + w \cdot d_{\text{tour}}(S'), \quad (6)$$

where $p_j(i)$ is the price of item i at store $s_j \in S'$, q_i is the quantity, $w = 0.5$ is the distance weight, and $d_{\text{tour}}(S')$ is the tour distance starting and ending at the user's location l_u , visiting each store in S' . Constraints include store proximity (within radius r), sufficient stock ($k_j(i) \geq q_i$), and payment method compatibility (m_u).

The ILP formulates the problem using binary variables:

- $x_s \in \{0, 1\}$: Indicates whether store $s \in S$ is selected ($x_s = 1$).
- $y_{i,s} \in \{0, 1\}$: Indicates whether item $i \in I$ is assigned to store $s \in S$ ($y_{i,s} = 1$).

The objective minimizes the total cost (tour distance is approximated post-solution due to TSP complexity):

$$\min \sum_{i \in I} \sum_{s \in S} p_s(i) \cdot q_i \cdot y_{i,s} \quad (7)$$

Subject to:

- **Item Assignment:** Each item is assigned to exactly one store:

$$\sum_{s \in S: k_s(i) \geq q_i, m_u \in M_s} y_{i,s} = 1 \quad \forall i \in I$$

- **Store Selection:** Items can only be assigned to selected stores:

$$y_{i,s} \leq x_s \quad \forall i \in I, s \in S : k_s(i) \geq q_i, m_u \in M_s$$

- **Minimum Stores:** At least one store is selected:

$$\sum_{s \in S} x_s \geq 1$$

The ILP is solved using Gurobi with a 4-second timeout, producing the optimal store set and item assignments. To generate up to $N = 20$ recommendations, the solver

iterates, adding constraints to exclude previous store combinations (e.g., $\sum_{s \in S'} x_s \leq |S'| - 1$). Post-solution, the tour distance is computed using a greedy TSP heuristic (nearest-neighbor from l_u to stores and back), and the score is calculated as $s = c + 0.5 \cdot d_{\text{tour}}$. The algorithm ensures feasibility by checking stock and payment method constraints during variable definition. Complete pseudocode can be found under the Appendix A.6 listing.

The time complexity depends on Gurobi’s branch-and-cut algorithm, typically exponential in the worst case ($O(2^{n+m})$ for n stores and m items) but efficient for small instances due to solver optimizations. The space complexity is $O(n \cdot m)$ for variables and constraints, plus $O(n^2)$ for the distance matrix.

4.9.2 Pipeline

The ILP is integrated into a pipeline that processes user input and delivers recommendations in real-time, adhering to the 4-second timeout. The pipeline, similar to that of Beam Search (Subsection 4.8.2) is tailored for exact optimization. The steps are:

1. **User Query:** Receive the user input $U = (I, q, l_u, r, m_u, N)$, specifying items, quantities, location, radius, payment method, and number of recommendations.
2. **Store Fetching:** Query a PostgreSQL database with PostGIS to retrieve stores within radius r of l_u , with sufficient stock for at least one item in I and supporting m_u . Limit to 150 stores, ordered by ascending item coverage to favor sparse stock and multi-store solutions.
3. **Distance Precomputation:** Fetch road distances from an OSRM (Open Street Routing Machine) backend server through the table API and cache in Redis. Construct a distance matrix for tour distance calculations.
4. **Data Preprocessing:** Initialize item IDs, quantities, and store-to-items mappings to define ILP variables and constraints efficiently.
5. **Algorithm Execution:** Solve the ILP using Gurobi with a 4-second timeout, generating up to N recommendations by excluding previous solutions.

6. **Post-Processing:** Sort the top N recommendations by score and format as (S', A) pairs, including store IDs, item assignments, total cost, tour distance, and score.
7. **Output:** Return recommendations to the user, logging runtime, best score, set size, and validity to the console.

The pipeline is visualized in Figure 4.5, a vertical waterfall diagram showing the flow from user input to recommendations. Redis caching and optimized database queries minimize latency, with ILP solving being the primary computational bottleneck. The pipeline processes user input through database queries, distance caching, and exact optimisation to deliver optimal recommendations within a 4-second timeout.

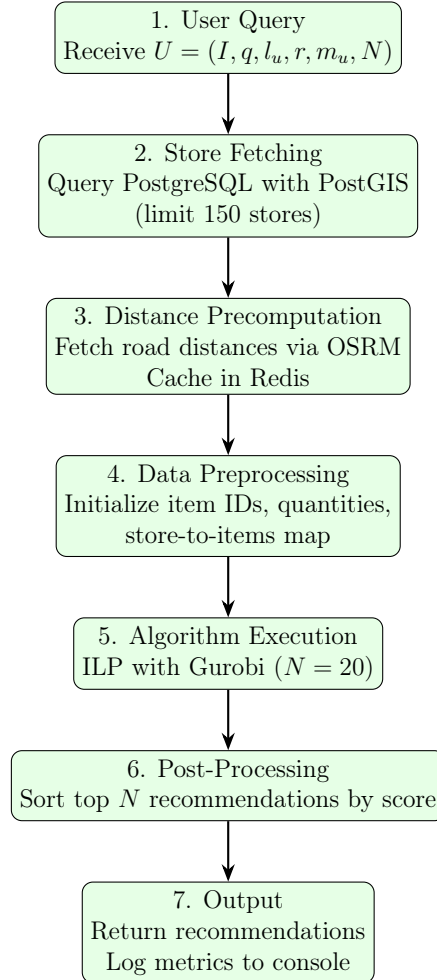


Figure 4.5: Vertical waterfall diagram of the Integer Linear Programming algorithm.

4.9.3 Optimization Steps

The ILP implementation is optimised for efficiency and multi-store solutions, building on lessons from Beam Search and Genetic Algorithm:

- **Sparse Stock Prioritisation:** The `fetch_stores` query orders stores by ascending item coverage (`ORDER BY item_coverage ASC`), favouring stores with partial stock to encourage multi-store solutions, mitigating the single-store bias observed in earlier algorithms (Subsection 4.8).
- **Reduced Distance Weight:** Setting $w = 0.5$ (down from 1.0) in the post-processed score reduces the tour distance penalty, making multi-store solutions competitive against single-store ones with shorter distances.
- **Distance Caching:** A precomputed distance matrix, stored in Redis with batched OSRM queries, minimizes network overhead for tour distance calculations.
- **Efficient Variable Definition:** Variables are only created for feasible item-store pairs (where stock and payment methods are compatible), reducing the ILP's size and solver runtime.
- **Gurobi Optimizations:** Gurobi's branch-and-cut algorithm leverages presolving, cutting planes, and parallel processing to accelerate solving, optimized for the 4-second timeout.

These optimizations ensure the ILP delivers optimal multi-store recommendations within the timeout, scalable to real-world inputs ($n = 150$, $m = 8$).

4.9.4 Pros and Cons

Pros:

- **Optimality:** ILP guarantees the optimal solution (within the timeout), providing a benchmark for heuristic algorithms like Beam Search and Genetic Algorithm.
- **Flexibility:** The ILP formulation easily incorporates additional constraints (e.g., maximum stores, brand preferences) by adding variables or constraints.

- **Real-World Integration:** The real-world version integrates seamlessly with PostgreSQL, Redis, and OSRM, handling complex datasets with unit conversions.
- **Precision:** Exact optimization ensures accurate cost minimization, critical for small instances where optimality is feasible.

Cons:

- **Scalability:** The exponential complexity limits performance for large instances ($n > 50$), often hitting the 4-second timeout.
- **Distance Approximation:** The post-processed greedy TSP heuristic overestimates tour distances, skewing the final score compared to the ILP’s cost optimization.
- **Resource Intensity:** Gurobi requires significant memory and computational resources, less practical for resource-constrained environments.
- **Timeout Dependency:** The 4-second timeout may yield suboptimal or incomplete solutions for complex inputs, reducing recommendation count.

4.9.5 Limitations and Future Improvements

The ILP, while optimal, has limitations that impact its practical use:

- **Scalability:** The exponential time complexity makes ILP impractical for large n or m , often exceeding the 4-second timeout.
- **Distance Handling:** The inability to include TSP in the ILP objective (due to non-linearity) relies on a post-processed greedy heuristic, reducing score accuracy.
- **Solver Dependency:** Reliance on Gurobi introduces licensing costs and compatibility issues, limiting deployment flexibility.
- **Timeout Sensitivity:** The fixed timeout may interrupt solving, returning fewer than N recommendations or suboptimal solutions.

One of the main limitations was the lack of libraries for library compatibility with other languages (Due to timeline constraints, conversion of those library API to Go was abandoned and considered as future work). Initial trials of this algorithm used Google’s OR-Tool library, written using the C++ language as a library for Python programs. Code for the attempt can be found under Appendix A.6.1.

Here are some future improvements that could enhance the ILP’s effectiveness:

- **Relaxation Techniques:** Use Linear Programming relaxation or Lagrangian relaxation to provide tighter bounds, speeding up the solver for large instances.
- **TSP Integration:** Approximate TSP within the ILP using linear constraints (e.g., Miller-Tucker-Zemlin formulation) to improve score accuracy.
- **Distributed Solving:** Leverage distributed ILP solvers (e.g., Gurobi’s distributed MIP) to handle large datasets ($n = 500$) across machines.
- **Heuristic Initialization:** Warm-start the ILP with solutions from Beam Search or Genetic Algorithm to reduce solving time.
- **Incremental Output:** Implement solution pooling in Gurobi to return partial results within the timeout, ensuring usability.

These enhancements would make ILP more practical for real-time retail applications. ILP algorithms are designed to give the optimal solution, meaning that taking multiple recommendations would go against their design decisions. This method is a good option when it comes to finding the most optimal stores to visit.

Summary and the selection of a suitable algorithm is included in the next section, which is dedicated to analysis. The next section discusses the route planning algorithm and its implementation.

4.10 Store Route Planning

The recommendation algorithms presented earlier generate a set of store combinations from which customers can select one to purchase goods. This subsection introduces an adaptive genetic algorithm designed to determine the optimal route for visiting the stores in the chosen combination, minimizing travel costs in terms of distance and time.

4.10.1 Problem Definition

The objective is to identify the most efficient route for a pre-selected store combination, factoring in travel costs defined by total distance and estimated travel time. The algorithm incorporates real-time traffic data to provide accurate time estimations, utilizing either the Haversine distance formula or the Google Maps API for distance calculations.

The algorithm comprises two primary components:

- **Genetic Algorithm (GA)**

The Genetic Algorithm employs evolutionary techniques—selection, crossover, mutation, and elitism—to explore the space of store combinations and their possible visiting orders. It identifies the sequence that minimizes the total travel cost, which combines distance and time metrics.

- **A* Algorithm**

The A* Algorithm computes the shortest path for a given store sequence, starting and ending at the customer's location. It leverages a heuristic-driven approach, integrating real-time traffic data and cached distance information to optimize route efficiency.

4.10.2 Methodology

This algorithm consists of two main components as earlier mentioned and employs a hybrid(adaptive) genetic algorithm with the following features.

- A* path finding algorithm for accurate travel time and distance estimations
- Traffic-aware fitness evaluation

- Adaptive genetic algorithm that adjust based on population diversity.

The algorithm iteratively evolves a population of potential solutions. Flow char of the algorithm is illustrated in the figure 4.6

4.10.3 A* algorithm

This algorithm is integrated into fitness evaluation by computing the shortest path for a given store order. Key features of the algorithm are listed below and the complete psuedocode can be found under Appendix B.1.

- Distance calculation -
Uses the haversine distance formula or Google Maps API. Precalculated distances are cached to prevent repeated work.
- Traffic data integration -
uses the traffic factor to modify the time spent traveling.
- Heuristic Function -
calculates the bare minimal distance for the stores that are left.

4.10.4 Genetic Algorithm

This algorithm is used to search the optimal combination and visiting order. It operates on a population of solutions, each representing a combination index and a permutation of store IDs. Key steps are highlighted below and the complete psuedocode can be found under Appendix B.2

- **Initialization**
Generate random solutions by shuffling store order
- **Fitness evaluation**
Uses A* algorithm to compute the total distance and time
- **Tournament selection**
Tournament selection - Eleminate unsatisfied solutions
- **Crossover**
Combines store orders

- **Mutation**

Sawps store orders or combinations randomly

- **Elitism**

Preserves the optimal solution to convergence.

4.10.5 Complexity Analysis

The complexity of the route planning algorithm is evaluated based on its two primary components: the A* pathfinding algorithm and the Genetic Algorithm (GA). This analysis provides insight into the time and space resources required for efficient execution, ensuring the algorithm's scalability and performance are well understood.

- **Time Complexity** - The A* pathfinding algorithm, employed to evaluate individual paths between stores, operates with a time complexity of $O(n)$ per path evaluation, where n represents the number of stores. This efficiency is achieved by assuming that distance data is cached, enabling rapid access during computations. In contrast, the Genetic Algorithm (GA) encompasses multiple operations per generation. The fitness evaluation process requires $O(p \cdot n)$ time, where p is the population size and n is the time to assess each solution's fitness. Tournament selection, a critical operation within the GA, incurs a time complexity of $O(p \cdot k \cdot n)$, with k denoting the tournament size. Additionally, crossover and mutation operations are executed in $O(p \cdot n)$ time. When considering all generations, the GA's overall time complexity becomes $O(g \cdot p \cdot n \cdot k)$, where g is the number of generations. Consequently, the total time complexity of the route planning algorithm is dominated by the GA, resulting in $O(g \cdot p \cdot n \cdot k)$.
- **Space Complexity** - In terms of space complexity, the A* pathfinding algorithm necessitates $O(n^2)$ space for caching distances between stores and an additional $O(n)$ space for storing individual paths. Meanwhile, the Genetic Algorithm requires $O(p \cdot n)$ space to maintain the population of solutions and $O(g)$ space to track statistics across generations. Combining these requirements, the overall space complexity of the algorithm is $O(n^2 + p \cdot n + g)$, reflecting the storage demands of both the A* and GA components.

4.10.6 Pros and Cons

The route planning algorithm offers several notable advantages that enhance its effectiveness and adaptability in real-world scenarios. One of its primary strengths is the combination of A*’s optimality for pathfinding with the Genetic Algorithm’s (GA) combinatorial search capabilities, allowing it to efficiently explore and optimize complex store sequences. The algorithm also supports real-time traffic data and accurate distance calculations, ensuring that the routes generated are practical and reflective of current conditions. Furthermore, the GA component provides flexibility and customization through adjustable parameters, enabling fine-tuning for specific use cases or constraints. Additionally, the use of caching significantly improves efficiency, particularly for repeated queries, by reducing redundant computations and speeding up response times.

Despite these strengths, the algorithm has certain limitations that must be considered. A key drawback is the non-deterministic nature of the results, as the GA’s stochastic processes can lead to variability in the solutions generated, potentially affecting consistency across different runs. The algorithm’s reliance on external APIs for real-world routing data introduces a dependency that could impact performance or availability if these services experience downtime or rate limiting. Moreover, the distance cache, while beneficial for efficiency, becomes memory-intensive as the number of stores n increases, potentially posing scalability challenges for very large datasets. These considerations highlight the need for careful parameter tuning and resource management when deploying the algorithm in production environments.

This subsection discussed the route planning algorithm; the evaluation details of this algorithm can be found in the Section 5.

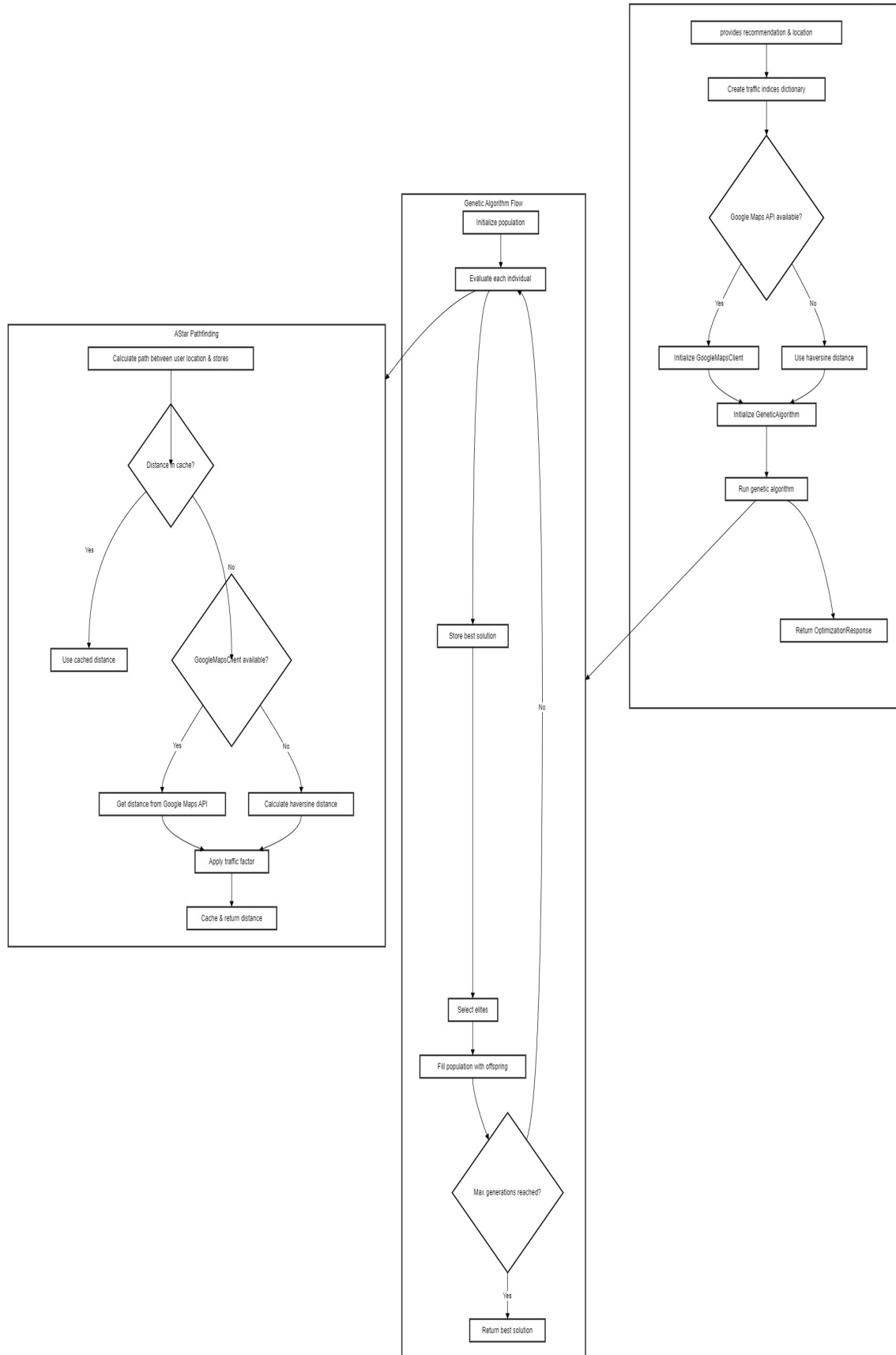


Figure 4.6: Main Flow Chart of the Route Planning Algorithm

4.11 Shopping list recommendation algorithm

To recommend a user a list of items to consider buying, an item recommendation algorithm was developed. A machine learning algorithm was deemed to be suitable for the task. As a dataset was not available, a simulated dataset was used to pick a suitable model for recommending items. An explanation of how the dataset was created and used to evaluate the models can be found below.

This study aimed to evaluate the performance of various collaborative filtering models for generating item recommendations, framed as predicting the likelihood of items appearing in users' shopping lists. The methodology encompassed the following stages:

4.11.1 Data Generation

For each test,

Simulated Users: A defined number of unique users were created, each with attributes such as a unique identifier, username, name, email, contact number, and password credentials. Timestamps for creation and updates were also generated.

Item Catalog: A pre-existing dataset of items (loaded from a csv file), containing unique item codes and names, served as the pool of available items.

Simulated Shopping Lists: User-item interactions were simulated to represent the items users might include in their shopping lists. This simulation was governed by several configurable parameters:

- **Number of Items per List:** Controlled the maximum number of unique items a user could include in a single shopping list.
- **Number of Lists per User:** Determined the total number of shopping lists simulated for each user.
- **Item List Persistence Threshold:** Introduced a temporal aspect to list creation. A proportion of the items from a user's previous simulated list was carried over to their subsequent list, mimicking users often considering similar or related items over time.

- **Randomness:** Random sampling was employed to select users, items, and quantities, with a fixed random seed for result reproducibility.
- **Store Association:** While store data was incorporated, the primary focus of the recommendation evaluation was on user-item relationships. Each simulated list entry was associated with a randomly chosen store identifier.

Shopping List Entries: Each simulated instance of an item appearing in a user’s shopping list generated an entry linking the user to a specific item and store, along with a quantity (representing, for example, the desired quantity of that item), a unique list identifier, and timestamps.

4.11.2 Recommendation Model Training and Evaluation

Collaborative Filtering Models: A range of collaborative filtering algorithms from the Surprise library were evaluated:

- **Singular Value Decomposition (SVD):** A matrix factorization method decomposing the user-item inclusion frequency matrix into lower-dimensional user and item latent factor matrices.

$$\mathbf{R} \approx \mathbf{U}\Sigma\mathbf{V}^T$$

- **SVD++:** An extension of SVD that incorporates implicit signals (the presence of items in lists) to enhance prediction accuracy.

$$\hat{r}_{ui} = \mu + b_u + b_i + \mathbf{q}_i^T \left(\mathbf{p}_u + \frac{1}{\sqrt{|N(u)|}} \sum_{j \in N(u)} \mathbf{y}_j \right)$$

- **Non-negative Matrix Factorization (NMF):** A matrix factorization technique with non-negativity constraints on the factor matrices.

$$\mathbf{R} \approx \mathbf{W}\mathbf{H}^T$$

- **KNN-based Algorithms (KNNBasic, KNNWithMeans, KNNWithZScore, KNNBaseline):**

Neighborhood-based methods predicting item inclusion based on similarities between users or items. Variations include mean centering, Z-score normalization, and baseline estimates.

$$\hat{r}_{ui} = \frac{\sum_{v \in N^k(u,i)} \text{sim}(u, v) \cdot r_{vi}}{\sum_{v \in N^k(u,i)} |\text{sim}(u, v)|} \quad \text{or} \quad \hat{r}_{ui} = \frac{\sum_{j \in N^k(i,u)} \text{sim}(i, j) \cdot r_{uj}}{\sum_{j \in N^k(i,u)} |\text{sim}(i, j)|}$$

- **Co-clustering:** A technique that simultaneously groups users and items based on their co-occurrence in shopping lists.
- **NormalPredictor:** A non-personalized baseline model predicting inclusion randomly based on overall item inclusion frequencies.

Data Preparation for Surprise: The generated shopping list data (user ID, item ID, and quantity) was formatted into a Surprise `Dataset` object. The quantity was used as the interaction strength for model training.

Train-Test Split: The data was partitioned into training and testing sets to assess the models' ability to generalize to unseen user-item relationships.

Model Training: Each selected model was trained on the training dataset.

Recommendation Generation: For each user, the trained models predicted the expected quantity for items not yet appearing in their lists. The top N items with the highest predicted quantities were considered recommendations.

Evaluation Metrics: The quality of recommendations was evaluated using precision, recall, and F1-score.

- **Relevant Items:** Items considered "relevant" for a user were those appearing in their simulated lists with a frequency exceeding a threshold determined by the total number of lists per user and a "relevancy threshold."
- **Precision:** The proportion of recommended items that were actually relevant to the user.

$$\text{Precision} = \frac{|\{\text{Recommended Items}\} \cap \{\text{Relevant Items}\}|}{|\{\text{Recommended Items}\}|}$$

- **Recall:** The proportion of relevant items that were successfully recommended

to the user.

$$\text{Recall} = \frac{|\{\text{Recommended Items}\} \cap \{\text{Relevant Items}\}|}{|\{\text{Relevant Items}\}|}$$

- **F1-Score:** The harmonic mean of precision and recall.

$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Averaging Metrics: Precision, recall, and F1-score were calculated for each user, and the average across all users was computed for each model and parameter configuration.

4.11.3 Parameter Exploration

A systematic exploration of different values for key simulation parameters was conducted:

- Number of Items per List
- Number of Lists per User
- Item List Persistence Threshold
- Relevancy Threshold

All selected recommendation models were trained and evaluated for each combination of these parameters to observe their impact on model performance.

4.12 Discount Engine

As discussed in the previous section, taking the diversity and complexity of discount rules, this system adopts a rule-based method with a specific syntax to define discounts. This model has its advantages and disadvantages. It is a language like SQL, but for describing discounts and their applicable conditions. This language allows users to specify conditions and actions for applying discounts to products or carts, using a structured format parsed by a Go-based engine. The rules are defined as strings and each rule consists one or two conditions, optionally connected by a logical operator, followed by an action. This engine is called, DCDQL(Discount Condition Definition

Query Language). Syntax definition of the discount rule language can be found under Appendix C.1 in the form of context-free grammar (CFG).

Below includes rules defined using this language with explanatory descriptions.

1. `product_id IN [1, 2, 3] THEN`

`product_percentage 10`

- *Description:* Applies a 10% discount to products with IDs 1, 2, or 3.

2. `min_cart_price > 500 THEN`

`cart_percentage 8`

- *Description:* Applies an 8% discount to the cart if the total exceeds 500.

3. `product_id IN [4, 5] AND category_id = electronics THEN`

`product_flat_amount 20`

- *Description:* Applies a \$20 discount to products with IDs 4 or 5 in the `electronics` category.

4. `purchase_quantity >= 5 THEN`

`cart_flat_amount 50`

- *Description:* Applies a \$50 discount to the cart if a product's requested quantity is at least 5.

This discount engine has three main parts. Lexer, parser and an evaluator. Lexer is in charge of tokenizing the strings and creating strings. The parser is in charge of enforcing the language syntax and is in charge of working on the tokens. Then, finally, the evaluator. The evaluator is the main entity in charge of calculating the actual discounts based on the rules defined by the clients. Complete psuedocode of the discount engine can be found under Appendix C.2.

To address the complexity that front-end users may encounter when defining rules, the system includes rule builders—user interface components designed to enable the dynamic creation of rules. Below are selected screenshots showcasing these rule builders. In this implementation of the discount engine, no evaluation was performed due to the inherent characteristics of the evaluator. The operational context of the evaluator

(a) Start the discount definition through the interface

(b) Defining a discount for the condition CATEGORY_ID equals 10000

Figure 4.7: Selected Screenshots of the discount definition interface - set 1

(a) Adding a second condition to the above condition using the AND connector

(b) Defining a action of the discount

Figure 4.8: Selected Screenshots of the discount definition interface - set 2

varies depending on the specific requirements of the application, resulting in changes to data types and runtime complexities. These variations are further influenced by

Easy Discount Rule Builder

Create discount rules in 3 simple steps!

1 Condition 2 Extra Condition 3 Discount

Step 3: Set the Discount

Discount type

Fixed amount off product

Discount amount

10

Back Generate Rule

Your Discount Rule

Rule (for system)

```
CATEGORY_ID = 10000 OR MIN_CART_PRICE = 1000 THEN
PRODUCT_FLAT_AMOUNT 10
```

Copy Rule

What it means

Apply a 10 off each product if the category equals 10000 or the cart total equals 1000.

(a) Final discount and it's English translation

Figure 4.9: Selected Screenshots of the discount definition interface - set 3

hardware constraints, which impact the engine's overall performance. Although the discount engine was developed, it was not adopted for practical use. This decision stems from its immaturity and inability to process discounts efficiently. A key limitation arises from the backtracking nature of the evaluation process. The algorithm first computes discounts for a subset of products before transitioning to evaluate cart-level discounts. However, the current design of the discount engine struggles to manage these scenarios effectively, often leading to redundant reevaluation of previously assessed cases. Discounts are not treated as a primary factor in the recommendation process. Instead, they are included in the output solely to inform the user of potential discounts that may be available at a given store. This approach ensures that while discounts are acknowledged, they do not drive the core recommendation logic.

4.13 Best Practices for Standardisation

The main idea was to propose guidelines for the APIs associated with the store locator application. The following are some examples of the guidelines defined in the system.

System uses JSON data formats to communicate with the outside world, while using protocol buffers to establish inter-server communication. This is done in order to reduce latency in the server-to-server communication. And JSON is used due to its ease of use, readability and understandability. The system uses HTTP status codes to indicate the status of each request to the client. In the process to increase the quality of the data, it was decided not to get the users enter new products to the system, the product list is maintained as a separate entity with the following fields.

```
{
  "productName": "Sliced Sourdough Bread",
  "code": "00151733",
  "brand": "Trader Joe's",
  "description": "Sliced bread for all",
  "brandTags": "trader-joe-s",
  "category": {
    "category": "Plant-based foods and beverages",
  },
  "labels": "Organic,EU Organic,FR-BIO-01",
  "imageUrl": "http://www.image-url.com/slice-bread",
  "productQuantity": "36 slices - 0.8kg",
  "servingSize": "1 slice (52 g)",
  "unitOfMeasure": "pcs"
}
```

Each product must have information about the fields, otherwise, it could lead to inconsistencies. Under each service for each of the routes that are implemented and exposed to the outside world, there must be a descriptive prefix in the URL path to inform the service being invoked. For example,

Standard: Use RESTful APIs with OpenAPI 3.0 specifications.

GET /products: Retrieve product details.

GET /inventory: Retrieve stock levels by store.

POST /inventory/update: Update stock quantities.

Authentication: Use OAuth 2.0 for secure access.

Response Format: JSON, with standard error codes (e.g., 200, 404).

Each store in the system must have the following data.

```
{
  "name": "Cop-City Kirimatiyana",
  "username": "Cop-City Stores",
  "email": "cop_city_kirimatiyana_a1@email.org",
  "address": "No, 34/1, Kirimatiyana Junction, Gampaha",
  "contactNumber": "+943596171048",
  "webUrl": "www.copcity-kirim.com",
  "locationPoint": {
    "latitude": 7.3544387,
    "longitude": 79.8857443
  }
}
```

This would ensure the consistency, likewise, there is a standard defined for each and every API endpoint that the system provides, and versioning will be done to provide backwards compatibility in case of an update. Following are some of the guidelines defined as action items regarding maintaining the quality of the APIs.

- Governance
 - Committee: Form a Retail Data Standards Committee to oversee updates.
 - Review Cycle: Update standards annually or as needed.
 - Feedback: Collect input via a dedicated portal or email.
- Implementation Support - To prevent users from deviating from the standard
 - Tools: Provide a JSON schema validator and API testing tool.
 - Documentation: Include detailed guides and FAQs.
 - Training: Offer webinars for store managers.
 - Support: Establish a help desk for technical assistance.

In this project, the following data sources were used to generate test data.

- Location data of stores and users:
 - Geofabrik
 - Planet OSM
 - Overpass API

Which are associated with the open source project -
OpenStreetMap(*OpenStreetMap on GitHub* 2025) project.
- Product information:
 - OpenFoodFacts(*Open Food Facts* 2025) - Another open source project dedicated to maintaining and accumulating product data from all over the globe.

Above information can be summarized as follows,

Table 4.1: Store Locator API and data collection Guidelines

Aspect	Store Locator Guidelines
Data Format	JSON
API Structure	RESTful APIs, OpenAPI 3.0
Collection Method	Web portal, API, CSV uploads
Governance	Retail Data Standards Committee
Tools Provided	Schema validator, API testing tool
Adoption Strategy	Incentives, training

The next section will present the evaluation results of the algorithms and the recommendation model.

5 Results and Analysis

This section will present the results and the evaluation of the results collected. Each subsection included in this section will have a similar anatomy, including the evaluation methodology followed, dataset generation, metrics used, and the results and analysis.

5.1 Evaluation of Route Planning Algorithm

This subsection provides a detailed description of the evaluation process for the implementation of the store route planning algorithm outlined in Subsection 4.10. It begins by explaining the evaluation methodology, then delves into performance metrics, followed by dataset generation, parameter configuration, and finally, the presentation of results and analysis based on the gathered data.

5.1.1 Evaluation Methodology

Assessing the algorithm’s efficacy, efficiency, and resilience under a range of situations and parameter setups is the goal of the evaluation. To examine the algorithm’s performance from several angles, the methodology employs an organized approach.

5.1.2 Performance Metrics

The performance metrics employed to evaluate the route planning algorithm are organized into three distinct categories: Solution Quality, Algorithm Efficiency, and Algorithm Robustness. Solution Quality metrics assess the practical effectiveness of the generated routes by examining key factors such as total travel distance, estimated journey time, and cost savings, all of which are critical for ensuring real-world applicability. Algorithm Efficiency metrics focus on the computational performance of the algorithm, measuring aspects such as execution time and convergence rate to verify its suitability for real-time applications. Algorithm Robustness metrics, in turn, evaluate the reliability and consistency of the algorithm, ensuring stable results across multiple runs and maintaining diversity in solutions to avoid suboptimal outcomes.

Solution Quality

- Total distance (meters): The total travel distance of the optimized route

- Total time estimate (seconds): The estimated time to complete the journey
- Average distance per store: Total distance divided by number of stores visited
- Average time per store: Total time divided by number of stores visited
- Estimated savings (%): Percentage improvement over baseline routes

Algorithm Efficiency

- Execution time (seconds): Time required to complete the optimization
- Convergence generation: Generation at which the algorithm stabilizes
- Convergence rate: Rate of improvement per generation

Algorithm Robustness

- Final diversity: Measure of genetic diversity in the final population
- Solution consistency: Coefficient of variation across multiple runs

5.1.3 Dataset Generation

Four separate datasets were created (full pseudocode used for generating data can be found under Appendix D.1 to reflect common use cases and edge situations in order to guarantee a comprehensive evaluation:

- **Small dataset:** 4 store combinations with 5 stores per combination
- **Medium dataset:** 10 combinations with 12 stores per combination
- **Large dataset:** 20 combinations with 20 stores per combination
- **Edge cases:**
 - **Single store dataset:** 1 combination with 1 store
 - **Empty dataset:** No valid combinations

5.1.4 Parameter Configurations

Several genetic algorithm parameter configurations were tested in the evaluation:

- Population size: 10, 50
- Generations: 5, 20, 50
- Mutation rate: 0.1, 0.2
- Crossover rate: 0.9 (fixed)
- Elite size: 2 (fixed)

5.1.5 Result and Analysis

The following are some evaluation results related to the Adaptive Genetic Algorithms
While the system details are as follows,

- **Host OS:** Windows 10 x86 64 (via WSL2)
- **Kernel:** 5.15.167.4-microsoft-standard-WSL2
- **Shell:** Bash 5.2.15
- **CPU:** 13th Gen Intel(R) Core(TM) i5-1335U
- **Memory:** 16 GB
- **Python Version:** Python 3.10.12

Solution Quality Analysis

The results for optimal route solutions across different datasets are presented in Table 5.1.

Table 5.1: Best Route Optimization Results by Dataset (TD - Total Distance, CG - Convergence Generation)

Dataset	Best TD (m)	Parameter Configuration	Execution Time (s)	CG
Small	39,069.89	pop=10, gen=5, mut=0.1	0.00	4
Medium	78,307.78	pop=10, gen=50, mut=0.2	0.04	49
Large	153,194.55	pop=50, gen=20, mut=0.2	0.02	9
Single	10,108.46	pop=10, gen=5, mut=0.1	0.00	0

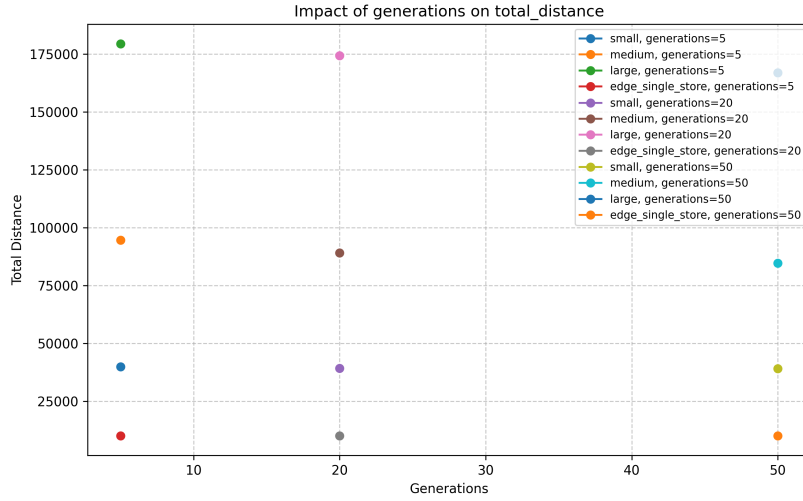


Figure 5.1: Total distance vs. generations across different datasets

Figure 5.1 illustrates the total distance values across different datasets and their convergence patterns.

Algorithmic Efficiency

Even for the largest dataset, the execution timings show remarkable efficiency, routinely falling below 0.11 seconds. Given the difficulty of the route optimization problem, this performance is noteworthy and points to a highly efficient implementation of the underlying data structures and genetic operators.

Table 5.2: Average Execution Time by Dataset and Configuration

Dataset	Pop=10, Gen=5, Mut=0.1	Pop=10, Gen=5, Mut=0.2
Small	0.01s	0.0s
Medium	0.01s	0.1s
Large	0.01s	0.02s
Single	0.00s	0.0s

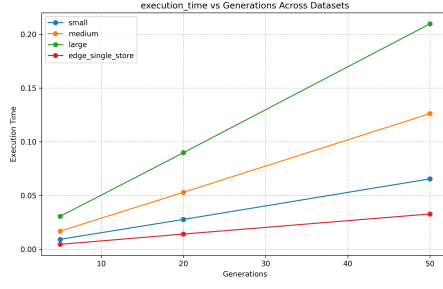


Figure 5.2: Execution time vs. dataset size

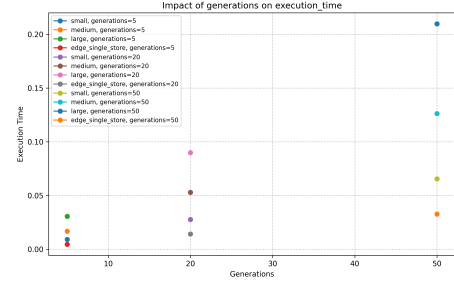


Figure 5.3: Execution time vs. Generations

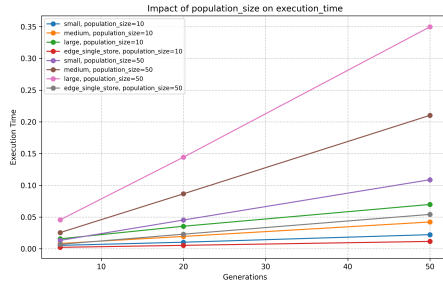


Figure 5.4: Execution time vs. Population

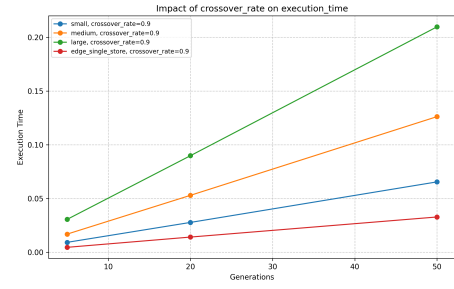


Figure 5.5: Execution time vs. Crossover

Figures starting from the Figure 5.2 to Figure 5.5, illustrates the comparison of execution time variations across multiple parameters: (a) dataset size, (b) number of generations, (c) population size, and (d) crossover rate.

Convergence Analysis

Information about how soon the algorithm finds optimal or nearly optimal solutions is provided by the convergence generation data:

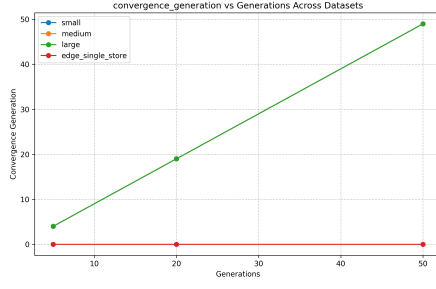


Figure 5.6: Convergence vs. dataset size

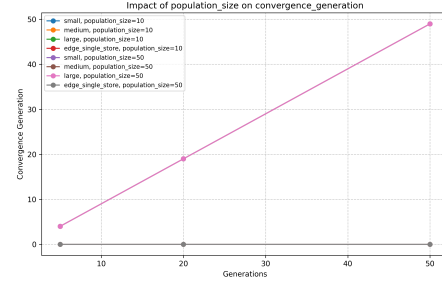


Figure 5.7: Convergence vs. Population size

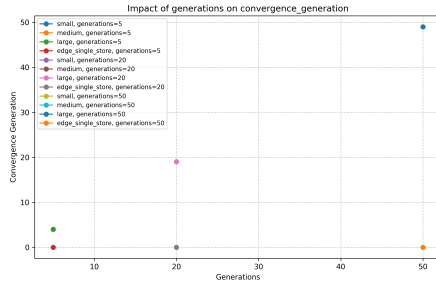


Figure 5.8: Convergence vs. Generation

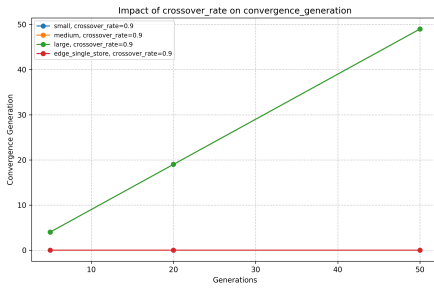


Figure 5.9: Convergence vs. Crossover

Figures starting from the Figure 5.6 to Figure 5.9, illustrates the comparison of convergence patterns across multiple parameters: (a) dataset size, (b) number of generations, (c) population size, and (d) crossover rate.

The evaluation of this route optimization algorithm based on a genetic algorithm with A* algorithm illustrates its efficiency and robustness. A variety of data sets with different parameters shows the comprehensive stat. Smaller dataset performs better with few configurations (ex: small population size, few generations). When it comes to large sets with more complexity, the algorithm performs better with different parameters (ex: large population). Algorithm shows the ability to handle edge cases as well.

5.2 Evaluation of Optimisation Algorithms

In this section, an analysis of the optimisation algorithms discussed in the previous sections will be conducted. The multi-store recommendation problem, as defined in Subsection 4.1.1, requires selecting a subset of stores and assigning items to minimise a score combining total cost and tour distance, subject to constraints like stock availability and store proximity. This subsection compares five algorithms designed for this problem: Branch and Bound (unoptimised), Branch and Bound with Beam Search, Parallel Branch and Bound, Integer Linear Programming (ILP), and Beam Search. Each algorithm is evaluated theoretically, focusing on optimality, scalability, complexity, flexibility, and practical considerations. A particular emphasis is placed on why Parallel Branch and Bound may be preferred, while assessing whether ILP or Beam Search could offer superior solutions in certain contexts. Table 5.9 summarises the comparison, followed by a discussion and a summary justifying the selection of Parallel Branch and Bound.

5.2.1 Algorithm Descriptions

This section revisits the algorithms from Subsection 4.3, offering detailed descriptions to enhance understanding of their mechanisms and applications.

Branch and Bound (Unoptimised) As explained before, these types of recursive algorithms systematically explore the solution space by constructing a tree where nodes represent partial store selections and item assignments. It uses a bounding function, typically based on cost and estimated tour distance, to prune branches that cannot result in better solutions than the current best. The algorithm guarantees optimality by evaluating all feasible combinations (this is very costly, for an input of the size 500 stores, 50 items and 2 payment methods), due to its exhaustive nature leads to high computational complexity, especially for large numbers of stores or items. Pruning relies on tight bounds, but without optimisations, it explores many redundant or suboptimal paths, making it impractical for large instances. With early pruning and the incorporation of concurrency, these types of algorithms could be performant as described in the latter part of this section.

Branch and Bound with Beam Search integrates Beam Search’s heuristic pruning into Branch and Bound. At each level of the search tree, it keeps only a fixed

number of promising nodes (the beam width, test for a BEAM_WIDTH=100 and BEAM_WIDTH=200), discarding others regardless of their potential (could remove more promising options, this is considered aggressive pruning). This reduces the number of explored paths, trading optimality for efficiency. The bounding function still guides pruning, but the beam width limits the solution space, potentially missing optimal solutions. This hybrid approach balances exploration and speed but depends heavily on the beam width parameter, which requires careful tuning to avoid this method from becoming an aggressive pruning method.

Parallel Branch and Bound enhances Branch and Bound by distributing the exploration of the search tree across multiple processors or threads (or machines, but in this context, distributed computing is not evaluated nor extensively looked into; rather, an overview was given in a previous section). Each processor explores a subset of branches independently, sharing the best solution found to tighten bounds globally. This preserves the optimality guarantee of unoptimized Branch and Bound while significantly reducing computation time through parallelization. The effectiveness depends on the number of processors and the overhead of communication, but it scales well for large instances, making it suitable for real-time applications with moderate to large datasets.

Integer Linear Programming (ILP) formulates the problem as a mathematical optimisation model with binary variables for store selection and item assignment. It minimises a cost-based objective (with tour distance approximated post-solution), subject to constraints ensuring each item is assigned to one store with sufficient stock. Other than most of the methods discussed so far, this algorithm is the easiest when it comes to adding new constraints or conditions (Just add a new variable to solve for). But due to its nature of mathematical modelling, as the number of variables to solve for increases, the run time and the space time start to increase. Solved using solvers like Gurobi, ILP guarantees optimality within a time limit but relies on branch-and-cut techniques, which can be computationally intensive. Its declarative nature allows easy constraint modifications, but scalability is limited for very large instances due to the exponential nature of integer programming.

Beam Search is a heuristic algorithm that explores the solution space level by level, maintaining a fixed number of partial solutions (beam width) at each step. It greedily

selects stores and assigns items based on a score combining cost and estimated tour distance, discarding less promising solutions. It will restrict the number of possible solutions to the beam width. Beam Search sacrifices optimality for speed, making it highly efficient for large instances. However, its reliance on local decisions and fixed beam width can lead to suboptimal solutions, especially if early choices exclude better paths.

5.2.2 Comparison

Table 5.3 and Table 5.4 compares the five algorithms across key theoretical dimensions: optimality, scalability, time complexity, space complexity, flexibility, and practical considerations. Each dimension is evaluated qualitatively to highlight trade-offs and inform the choice of Parallel Branch and Bound.

Table 5.3: Comparison of Branch & Bound Variants

Criterion	B&B (Unopt.)	B&B with Beam	Parallel B&B
Optimality	Guaranteed	Not guaranteed; depends on beam width	Guaranteed
Scalability	Poor; exponential growth	Moderate; limited by beam width	Good; scales with processors
Time Complexity	Exponential ($O(2^{n+m})$)	Reduced exponential ($O(b \cdot d)$)	Exponential, parallelized
Space Complexity	High; stores full tree	Moderate; stores beam	High; distributed across processors
Flexibility	Moderate; requires bound redesign	Moderate; tuning beam width	Moderate; same as unoptimized
Practical Considerations	Impractical for large instances	Sensitive to beam width tuning	Requires parallel hardware

n : # stores, m : # items, b : beam width, d : tree depth.

Table 5.4: Comparison of ILP and Beam Search

Criterion	ILP	Beam Search
Optimality	Guaranteed (within time limit)	Not guaranteed; local optima
Scalability	Moderate; solver-dependent	Excellent; linear with beam width
Time Complexity	Exponential (solver-dependent)	Linear in beam width ($O(b \cdot m)$)
Space Complexity	Moderate; solver manages memory	Low; stores beam
Flexibility	High; easy constraint addition	Low; fixed heuristic structure
Practical Considerations	Solver dependency, licensing	Fast but suboptimal

n : # stores, m : # items, b : beam width, d : tree depth.

The following are some evaluation results related to the branch and bound variants. Evaluations were conducted on, PostgreSQL DBMS, hosting 8,927 products, 948 stores, and 7,191,173 store-product mappings and Redis server, with 75.96 MB memory usage on a Docker container. While the system details are as follows,

- **Host OS:** Windows 10 x86_64 (via WSL2)
- **Linux Distro:** Debian GNU/Linux 12 (Bookworm)
- **Kernel:** 5.15.167.4-microsoft-standard-WSL2
- **Shell:** Bash 5.2.15
- **CPU:** Intel Core i7-11800H (8 cores / 16 threads)
- **Memory:** 8 GB (6.7 GB used during tests)
- **Docker:** Engine 28.0.1, Compose v2.33.1
- **Go Version:** go1.23.5
- **PostgreSQL:** Version 17.4 (Debian PGDG)

Table 5.5: Performance Timing Metrics for Each Algorithm

Algorithm	Avg Algo Duration (s)	Avg Query Duration (s)	Success Rate
beam_search	0.064	0.166	0.483
beam_search_set_limited	0.064	0.166	0.483
branch_and_bound	0.096	0.172	0.600
branch_and_bound_parallel	0.035	0.166	0.539
branch_and_bound_with_beam	0.091	0.166	0.112
genetic	0.058	0.172	0.282

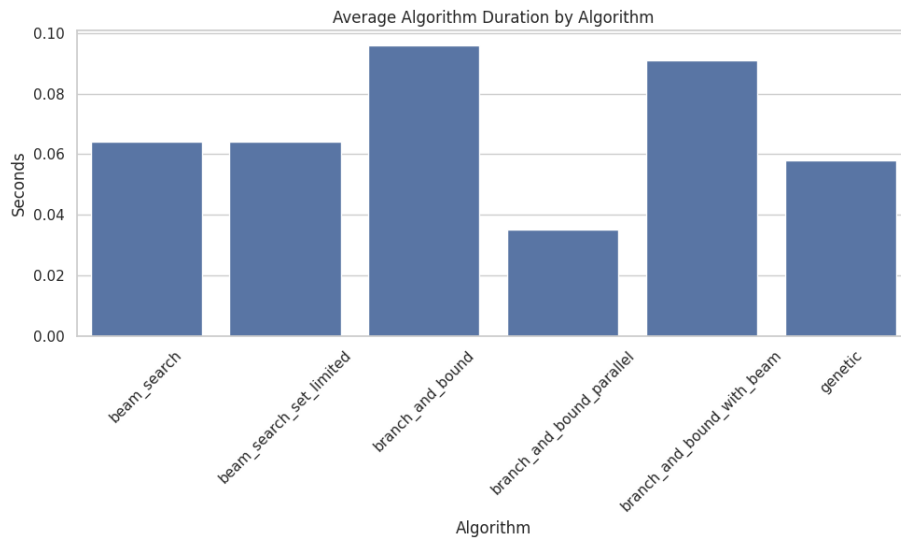


Figure 5.10: Recommendation Algorithm Average Durations

Table 5.6: Performance Quality Metrics for Each Algorithm

Algorithm	Avg Best Score	Avg Num Items	Avg Num Stores
beam_search	2328.44	21.24	13.03
beam_search_set_limited	2296.28	21.24	13.03
branch_and_bound	2267.76	22.05	13.44
branch_and_bound_parallel	2147.11	21.24	13.03
branch_and_bound_with_beam	279.26	21.24	13.17
genetic	829.65	22.05	13.44

Table 5.7: Detailed Statistics for Branch and Bound

Metric	Value
Avg Algo Duration (s)	0.096
Avg Query Duration (s)	0.172
Avg Best Score	2267.76
Success Rate	0.600
Avg Num Items	22.05
Avg Num Stores	13.44
Timeout Rate	1.000

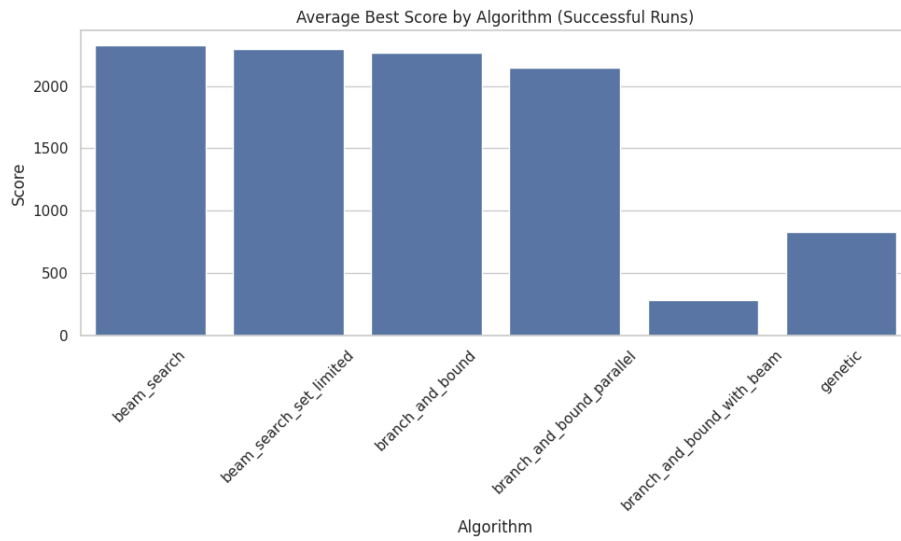


Figure 5.11: Recommendation Algorithm Average Best Scores

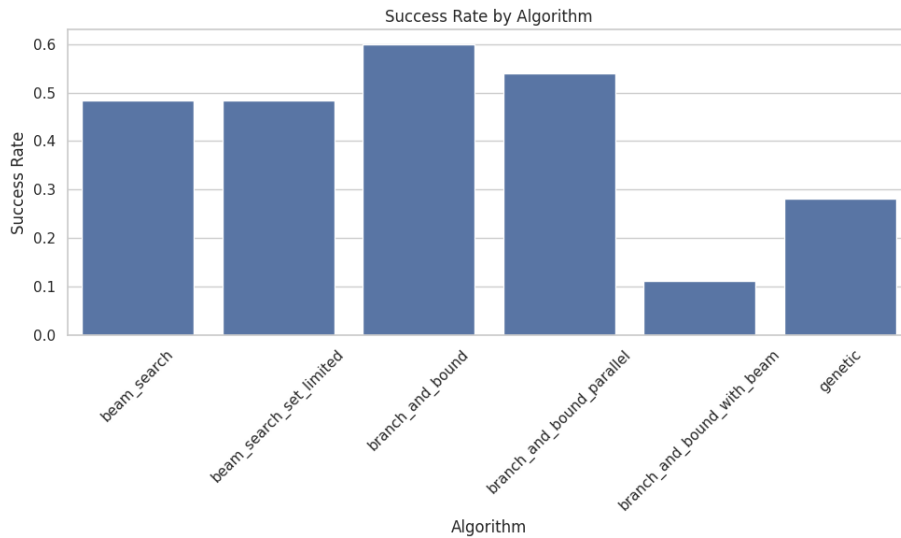


Figure 5.12: Recommendation Algorithm Average Success Rates

Practical Considerations: Unoptimized Branch and Bound is impractical for

Table 5.8: Branch and Bound Performance by Number of Items

Num Items	Avg Query Duration (s)	Avg Algo Duration (s)	Avg Num Recs	Avg Best Score
2	0.044	0.101	3.000	70.30
4	0.052	0.062	1.600	101.87
10	0.139	0.101	3.235	461.61
20	0.179	0.101	3.125	982.54
30	0.413	0.108	2.500	1550.85
40	0.235	0.102	5.000	3911.26
50	0.270	0.105	2.750	3042.94

Table 5.9: Evaluation Results of Recommendation Algorithms

Algorithm	Runtime Metrics		Recommendation Metrics		
	Avg Duration (s)	Avg Best Score	Success Rate	Avg Num Items	Avg Num Stores
beam_search	0.064	2328.44	0.483	21.24	13.03
beam_search_set_limited	0.064	2296.28	0.483	21.24	13.03
branch_and_bound	0.096	2267.76	0.600	22.05	13.44
branch_and_bound_parallel	0.035	2147.11	0.539	21.24	13.03
branch_and_bound_with_beam	0.091	279.26	0.112	21.24	13.17
genetic	0.058	829.65	0.282	22.05	13.44

large instances due to its computational resource requirements. Branch and Bound with Beam Search requires careful beam width tuning, and as the statistics shows the `branch_and_bound_with_beam` had the lowest success rate, meaning the algorithm yield infeasible as it's answers for larger datasets, which can be challenging in dynamic settings, as this algorithm could lead to infeasible states frequently. Parallel Branch and Bound demands parallel hardware but offers robust performance for real-time applications.

Why Choose Parallel Branch and Bound? Parallel Branch and Bound combines the optimality guarantee of unoptimized Branch and Bound with improved scalability through parallelisation. As you can see from the statistics this algorithm has the second highest success rate and it has the least duration, when compared with other algorithms. Unlike ILP, it avoids solver dependencies, making it more portable and cost-effective. Compared to Branch and Bound with Beam Search and Beam Search, it ensures optimal solutions without relying on heuristic pruning, which can miss better solutions. Its ability to leverage parallel hardware makes it suitable for real-time retail

applications with moderate to large datasets, balancing speed and quality. Beam Search is better for scenarios prioritising speed over optimality, especially with very large instances.

Is There a Better Solution? Among the compared algorithms, Parallel Branch and Bound is often seen as a good choice for its balance of optimality and scalability, but ILP (even though, the section doesn't evaluate this algorithm, only an explanation is given in the implementation stages due to the challenges faced) could be better when complex constraints (e.g., budget limits, store preferences) are needed, as its declarative capabilities simplify modifications. Beam Search is preferable in time-critical applications with large datasets where suboptimal solutions are acceptable. Branch and Bound with Beam Search offers a middle ground but is less reliable due to its sensitivity to beam width, and it's evident from the stats that the algorithm has a very low success rate without the incorporation of other optimisations. Unoptimised Branch and Bound is rarely practical due to its poor scalability.

5.2.3 Summary of Optimisation algorithms

Parallel Branch and Bound is the preferred algorithm for the multi-store recommendation problem due to its ability to guarantee optimal solutions and scale through parallelisation, high success rates and it's average best score is less compared (lower the better) to the successful algorithms (not the branch and bound with beam version and the genetic version). It outperforms unoptimized Branch and Bound by leveraging multiple processors, avoids the solver dependencies of ILP, while ILP excels in flexibility and Beam Search in efficiency, Parallel Branch and Bound strikes a balance, making it ideal for real-time retail applications requiring high-quality recommendations within practical time constraints. And this could be improve with a more efficient pruning metric in the branching state and the bounding states.

5.3 Evaluation of Recommendation Model

Next, the evaluation of the recommendation model will be done. The average precision, recall, and F1-score for each model and parameter setting were stored in CSV files. Visualisations were generated to analyse the relationships between simulation parameters, the number of recommendations (N), and performance metrics:

- Line plots: Illustrating the impact of the number of recommendations (N) on the average F1-score for each model.
- Heatmaps: Showing the relationship between pairs of simulation parameters (e.g., items per list vs. lists per user) on the average F1-score, separated by recommendation model.
- Box plots: Comparing the distribution of average F1-scores across different recommendation models.

The results of these experiments, detailed in the generated CSV files and visualizations, revealed the relative performance of the different recommendation models under the various simulated conditions. The following is a summary of the key findings with accompanying visualizations:

5.3.1 Model Performance Hierarchy:

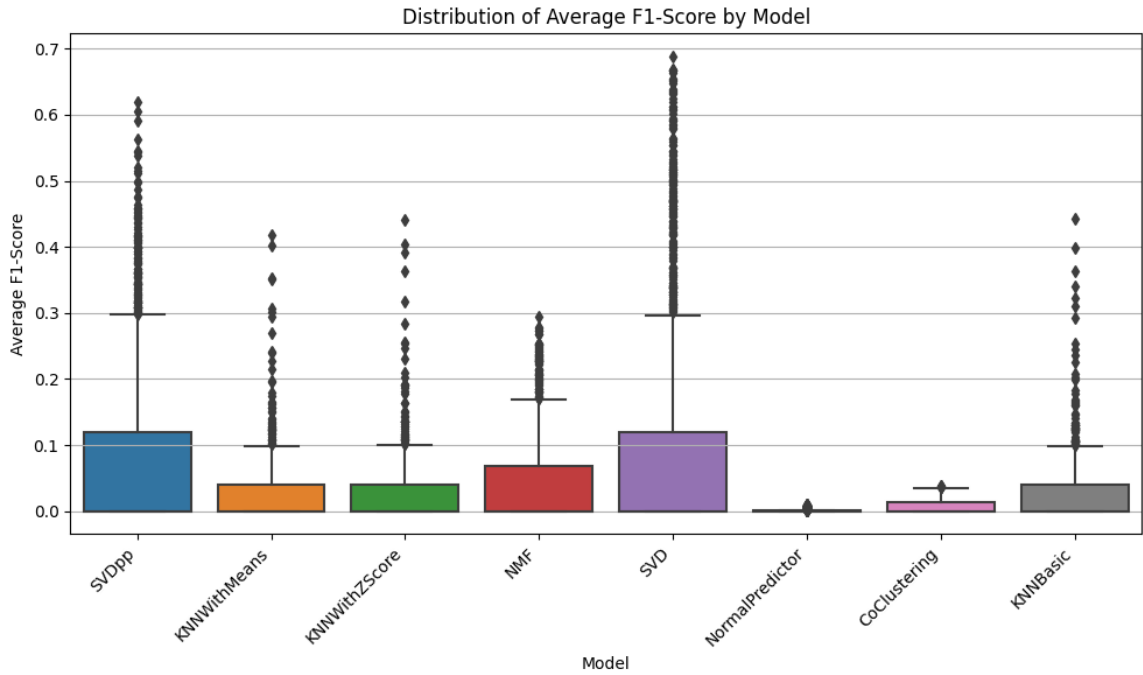


Figure 5.13: Distribution of average f1 score by model

As can be seen in Figure 5.13, the different recommendation models exhibited varying degrees of performance across the simulated conditions. The conditions were made to be the worst case scenario when it came to the parameters, which is why every model is not shown to have great performance in this distribution. However, when it comes

to normal scenarios, the models performed much better. The unfavourable performing results for all the models were kept in the evaluation to avoid chasing confirmation bias.

The baseline model, NormalPredictor had the lowest average F1-score. It failed to generate good recommendations for most users, and the other models performed better in almost every case giving it a score of 0.0 or close to it. Co-clustering was slightly better than NormalPredictor, but still not as good as the other models. The KNN-based models performed even better, but not to a satisfactory level. NMF was better than the other KNN-based models, but still not as good as the SVD-based models. SVD and SVD++ performed the best, with the highest average F1-scores and maintaining the best scores for normal scenarios.

A normal scenario would be one where there are a significant number of users, with each user having a number of lists with about 10 items in each list. These lists would also have a persistence threshold of around 50-90%. Meaning that the each list would retain that percentage of the items from the previous list. The persistence was done not by retaining a number of items for all lists and filling the gaps but by retaining from each subsequent lists and adding random items to fill the gaps. This would mean that the lists would have a similar number of items, but the lists would become diverse for each user.

The results from such a normal scenario for the best performing models (SVD and SVD++) can be seen in the figures Figure 5.14 and Figure 5.15 respectively.

The ultimate target for this model is to create a recommendation for a list containing at most 20 items. Store surveys suggested that this would be around the upper limit of the number of items a user would include in a shopping list. The SVD and SVD++ models performed well in this scenario but their characteristics were different as can be seen in the figures 5.14 and 5.15. The SVD++ model increases its recall as the recommended list goes, meaning that it will recommend more correct items as the list grows as opposed to SVD which plateaus to around 0.7. But the accuracy for SVD is much better than SVD++ on smaller lists which are the most common scenarios. The very low recall of SVD++ on normal scenarios makes it a poor choice for this task. The F1-score on average for SVD++ on the worst case scenarios is better than SVD. But as can be seen here the overall best choice for this recommendation problem based

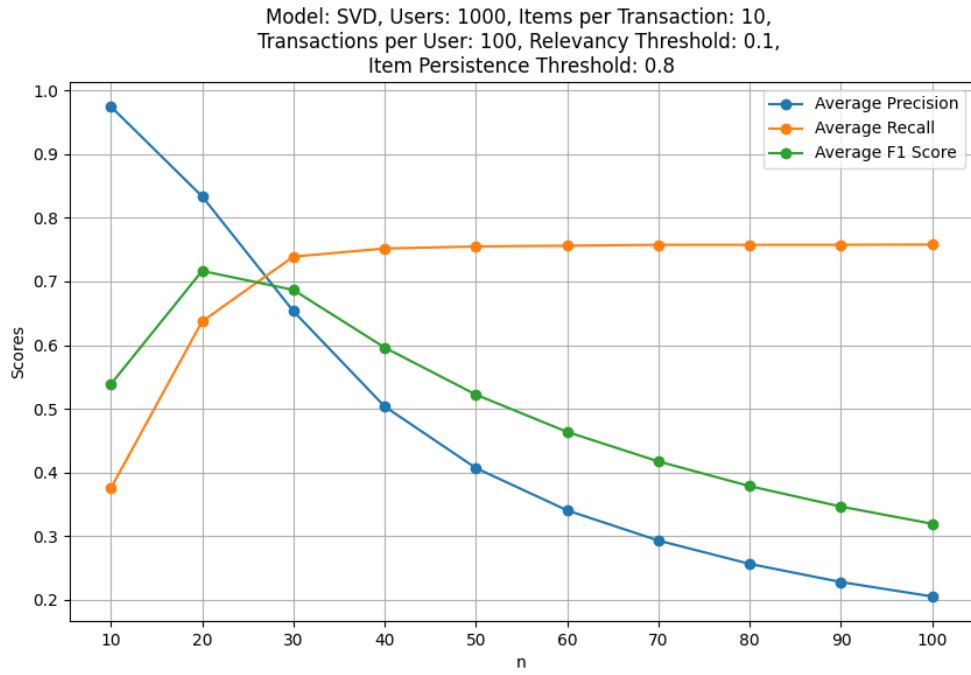


Figure 5.14: Scores for a normal scenario by SVD

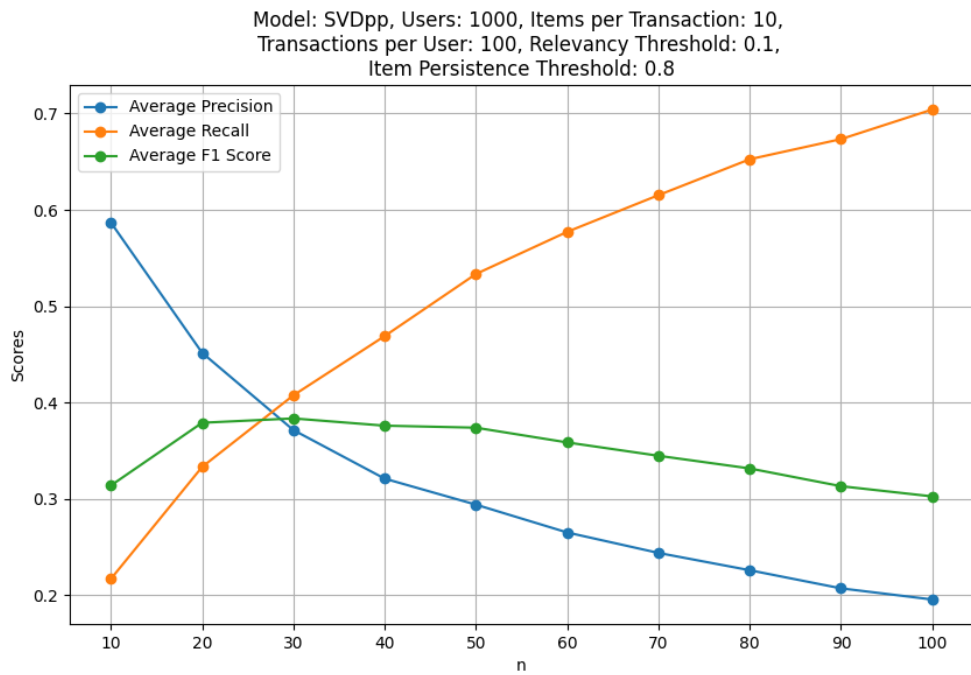


Figure 5.15: Scores for a normal scenario by SVD++

on F1-score is SVD out of all the models tested.

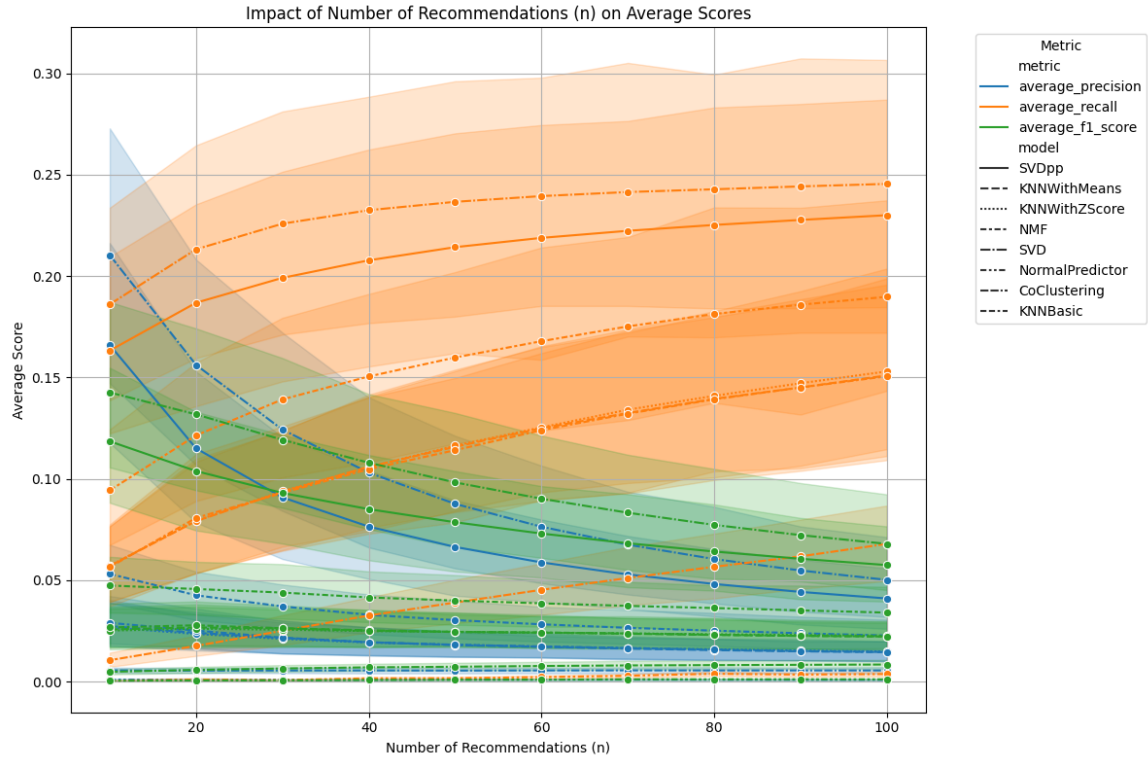


Figure 5.16: Impact of number of recommendations (n) on average scores

5.3.2 Impact of Recommendation Count (N):

The number of recommendations (N) is the number of items recommended to a user in a list. The target as mentioned previously is to recommend at most 20 items. The results from this experiment can be seen in Figure 5.16. The top performing models from above demonstrated almost identical behavior in in this regard as can be seen in the figure. The average recall increases as the number of recommendations increases. and the accuracy drops as the number of recommendations increases. The average F1-score for the models sometimes increases from 10 to 20 but in general it drops as the number of recommendations increases. This was used to see whether any model would excel at recommending a large number of items.

5.3.3 Influence of Simulation Parameters:

The simulation parameters that were used to generate the data were analyzed to see how they affect the performance of the models. The results from this experiment can be seen in following figures. The F1-score is used as the metric to highlight with the changing of the parameters as it is a good indicator of the model's performance. The scores are averaged across all the users and the models. The F1-scores may seem particularly low for these evaluations but this is because the evaluations are averaged for the worst case scenarios. The results from the normal scenarios are still satisfactory. This study is not meant to be a proof of the best model for this task but rather to highlight the performance of different models under different scenarios.

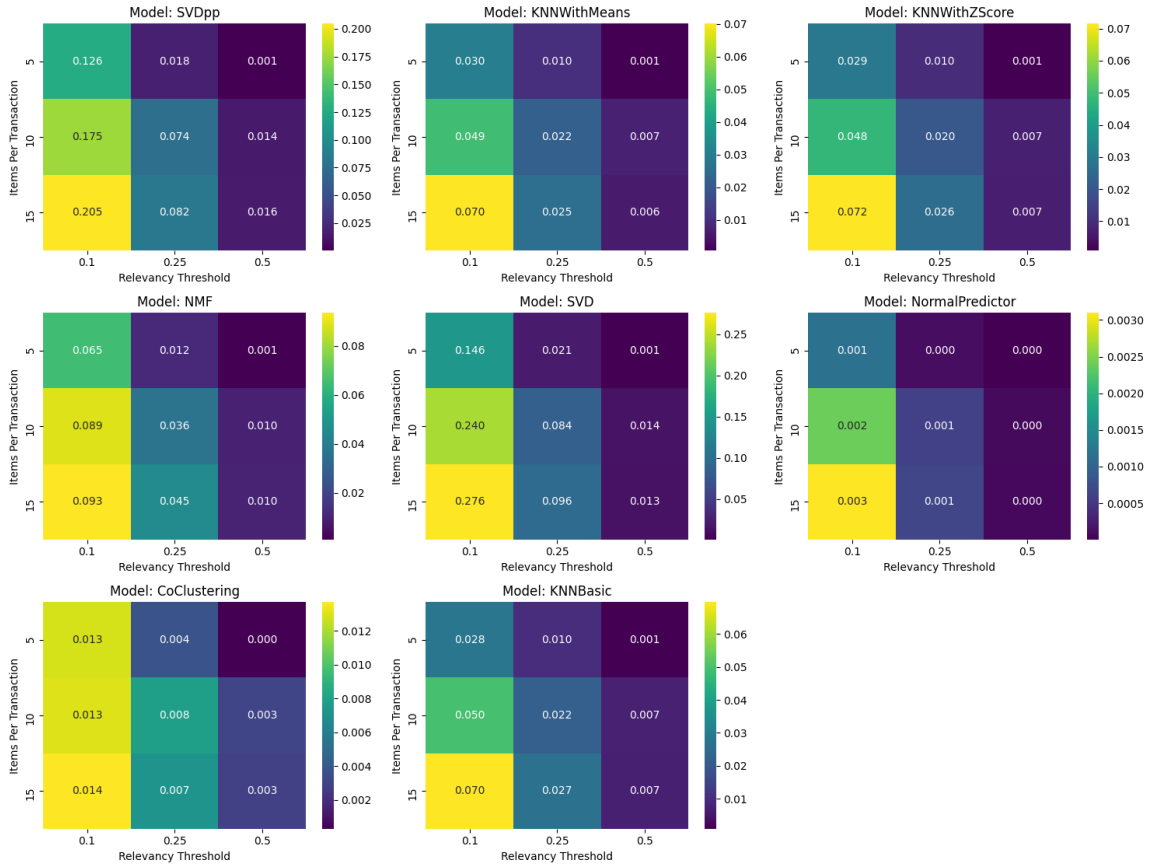


Figure 5.17: Average F1-score vs Items per List vs Relevancy Threshold

Here in figure 5.17 the F1-score is plotted against the number of items per list and the relevancy threshold. The Items per list seemed to have less of an impact than the relevancy threshold. This is because the relevancy threshold is a threshold for the number of items that a user would include an item in their shopping lists. With the

lists being as random as they are, only very low relevancy thresholds are useful in this test. In the real world, higher numbers may also work. But this shows that models like SVD and SVD++ can still find the most relevant items much better than the other models.

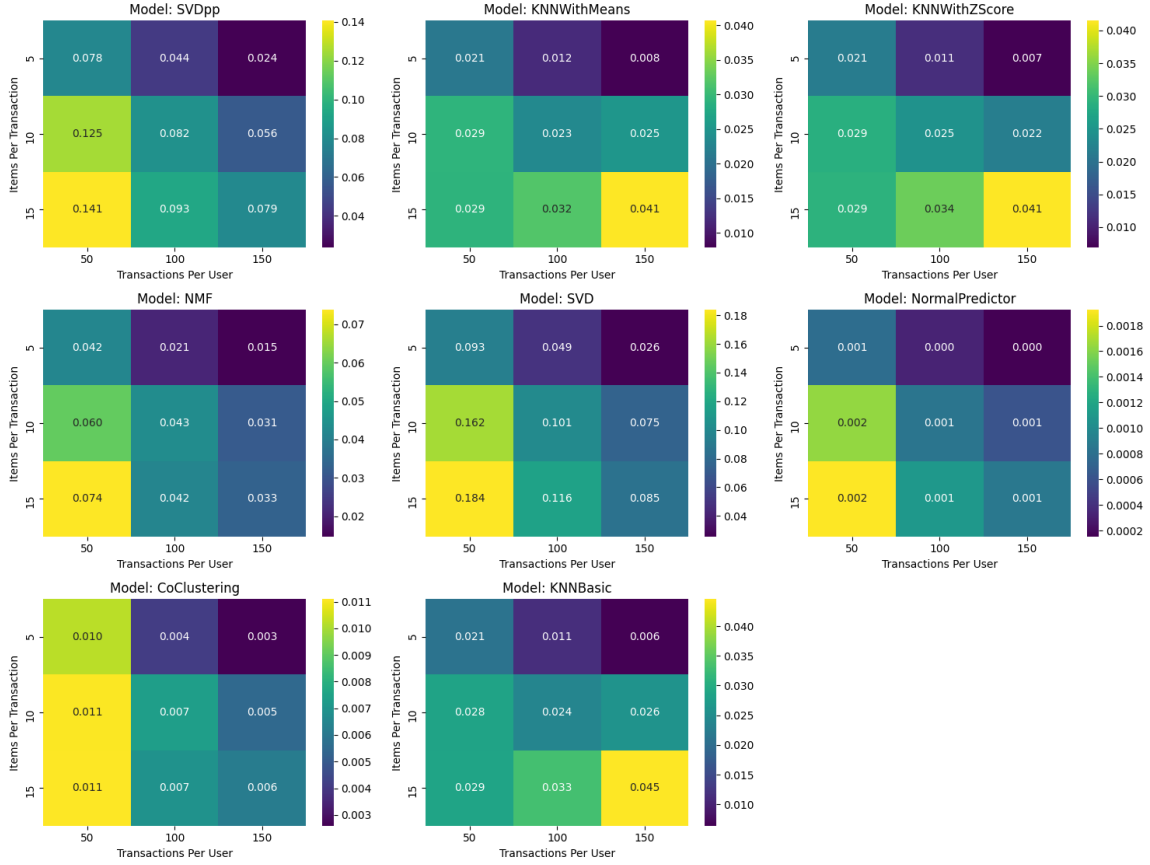


Figure 5.18: Average F1-score vs Items per List vs Transactions per User

The randomness in list generation is apparent in this analysis as the F1-score degrades when the transactions per user is increased. A higher score could have been achieved if the item persistence behaviour was changed to repeat a fraction of items for a user's lists without randomness, but that would have been too much of a beneficial scenario for these models and would have made the results less representative of the worse cases. The results from the normal scenarios are still satisfactory. Chasing confirmation bias is not a good way to evaluate the performance of a recommender system.

Here it can be seen that without list persistence and a low relevancy threshold, the F1-score is very low for all models. The models perform well only when there is a semblance of a pattern to the data as expected. It can be observed that relevancy threshold is less of a factor when the list persistence is high for some models like

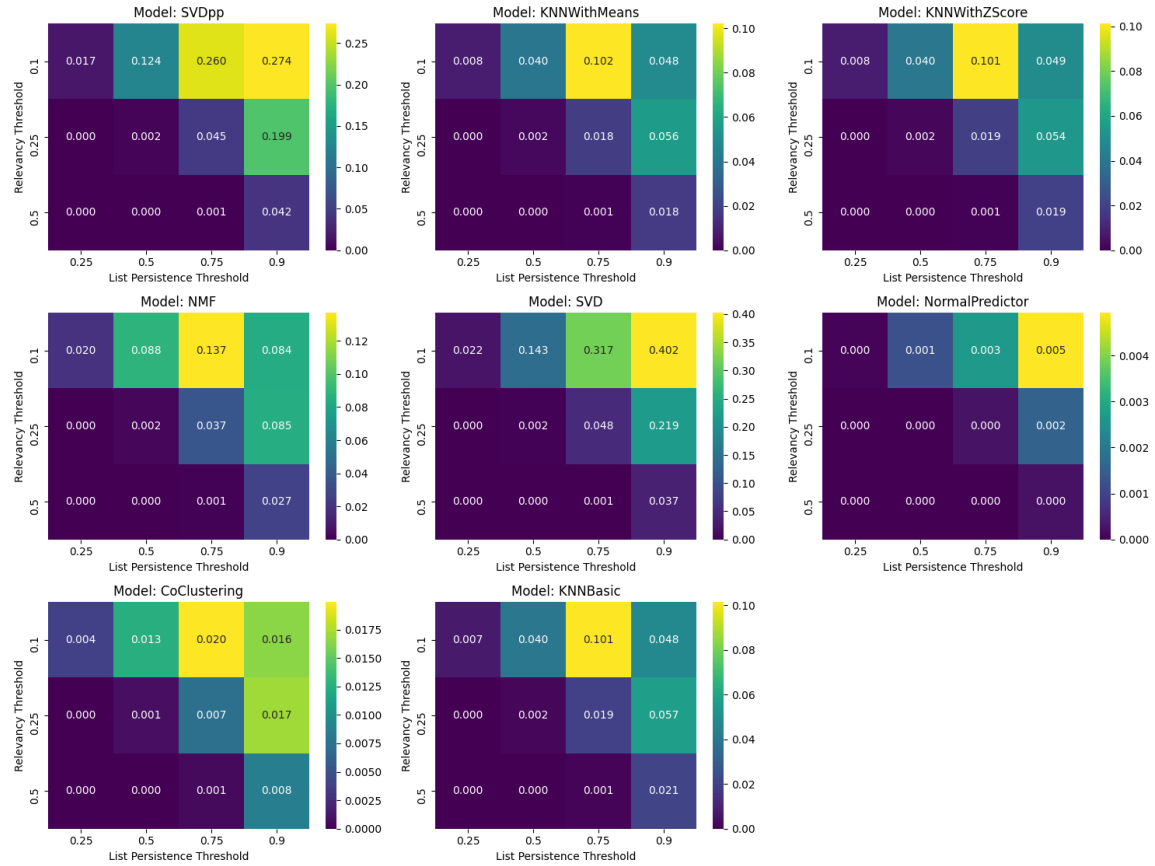


Figure 5.19: Average F1-score vs Relevancy Threshold vs List Persistence Threshold

NMF, the KNN-based models and Co-clustering. SVD based models scale with the relevancy threshold and the list persistence threshold. But the F1-scores show that even when affected, SVD and SVD++ perform better than the other models.

The KNN-based models are less affected by the number of lists per user but the persistence threshold still affects them greatly here. The SVD and SVD++ models perform better when the lists are persisted for longer periods of time. Under-performing sections of SVD and SVD++ still contain exceptional scores compared to the others. SVD being better still.

In conclusion for this problem, with the above results and analysis, the recommendation model for this problem was evaluated under a variety of simulated scenarios to determine its suitability for suggesting shopping list items. A range of collaborative filtering models was tested using data that aimed to replicate real user behavior, including temporal persistence and randomness. Among these, SVD emerged as the most balanced performer, particularly in typical scenarios where users have moderately persistent lists with a manageable number of items. While SVD++ showed strength in

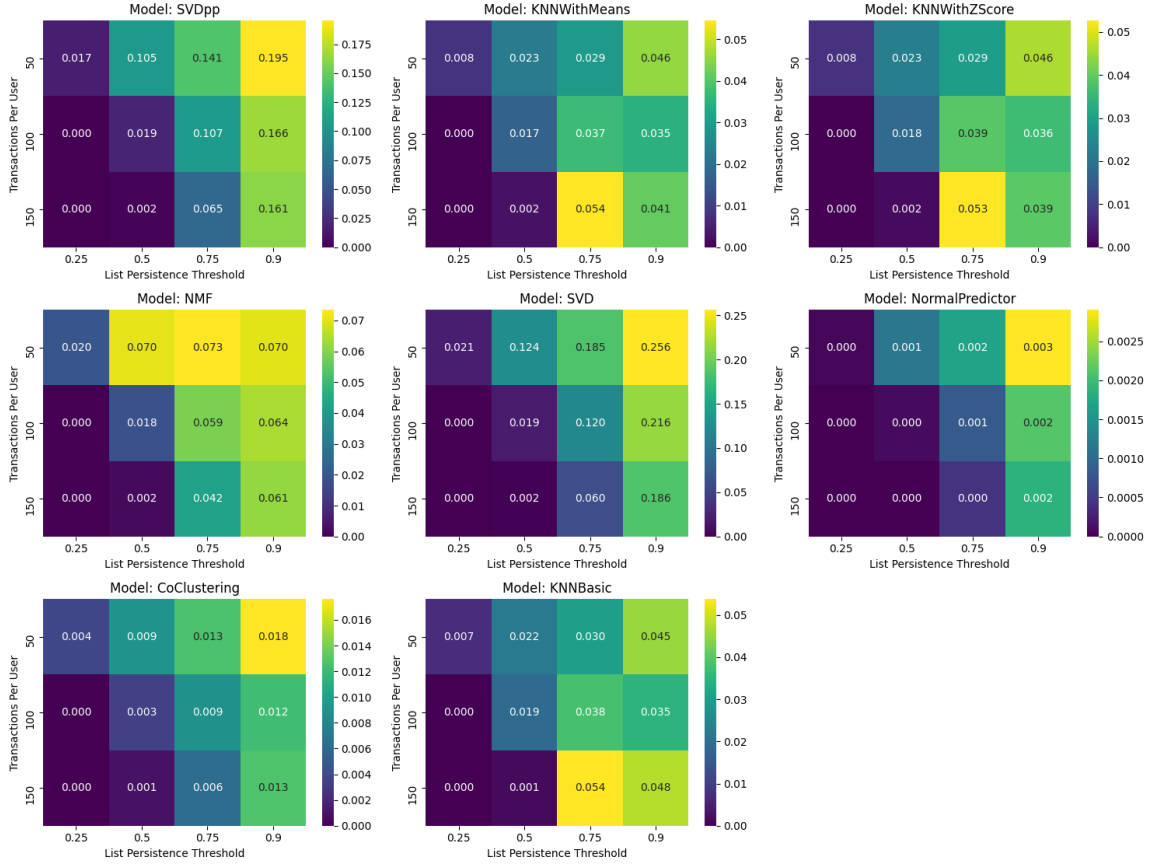


Figure 5.20: Average F1-score vs Lists per User vs List Persistence Threshold

recall when longer lists were recommended, its performance on smaller, more realistic lists did not justify its use over SVD.

The results reflected that model performance was strongly dependent on the characteristics of the input data, especially persistence and relevancy thresholds. Even under worst-case configurations, SVD-based models remained the most reliable, avoiding major performance drops seen in other approaches. These findings supported the selection of SVD as the core model for generating item recommendations in the system, with the potential to adapt further as more real-world usage data becomes available.

When it comes to future work relating to this model and the problem at large, it is a fact that the recommendation algorithm was developed and evaluated using simulated data, which, while diverse and configurable, does not capture the full complexity of real-world user behaviour. Moving forward, the next step would be to deploy the model within the live system and gather actual user, interactions to retrain and fine-tune its predictions. This would allow the model to adapt to real preferences and shopping habits over time.

Additionally, efforts will be made to incorporate context-aware features that were highlighted in the user survey, such as time of day and seasonal patterns, which may further improve recommendation quality.

Finally, user feedback mechanisms should be added to allow refinement of suggestions over time and help reduce noise from unusual or one-time purchases. This feedback loop could serve as a valuable source of implicit and explicit signals for future model improvements.

5.4 Qualitative Survey Results

Apart from the systematic evaluation, a qualitative survey was conducted by visiting local stores to gain insights into the practical concerns associated with a project like this. The rest of this section is dedicated to explaining the results of the qualitative survey results and insights. The insights identified were as follows,

- The survey discovered that customers have a tendency not to explore more than 3-4 stores to get the desired products. This heuristic is actually used when implementing the branch and bound parallel algorithm. This algorithm, for most cases, tries to limit the number of stores in a recommendation to 3-4. And due to this, this algorithm can handle a vast number of stores as well. It is evident when looking at the number of average stores considered to make recommendations.
- The survey identified hectic business hours to be 6 A.M. - 9 A.M. in the morning and, 11.30 P.M. - 2.40 P.M. due to lunch hour and due to where the stores are situated, many stores that are near to public infrastructure (like schools, for example) confirmed this, and from 5.40 P.M. to about 7.00 P.M. were recorded as busy hours for the evening. Queue sizes are most likely to be at their longest near or between those hours. But some stores (about 1km or more away from a town) confirmed that the queues consisted of less than 5-7 people, within the said periods.
- Querying about stock supply frequencies revealed that many of the stores frequently resupply perishable/consumable items like dairy products (Milk packets, Nestle products and other related products), Tea packets and soap. Unless for seasonal holidays where almost all items need to be restocked more frequently.

- The next questions were about their point of sale systems, and what sort of methods they used to keep track of daily sales. Many of the stores (smaller stores) didn't use POS systems or other methods other than pen and paper to keep track of their daily sales. Most of the mini supermarkets had POS systems and didn't like the idea of integrating some external software into them. But some stores liked the idea if it helps with better management of their products and sales in general. Some liked the idea of giving IoT devices or some sort of embedded devices to communicate with our servers, while some wanted to use their mobile phone, which is ideal, due to the use of bar-code scanning to keep track of stocks. Due to the diversity of data and differences between communication method.
- Next question was regarding the discounts provided by the stores and the payment methods. Some of the mini-supermarkets(3) provided card payment options, while many of the mini or small stores only accepted cash payments. Many of the stores didn't have discount offers, other than BOGO(buy one get one free, especially for dairy products like yogurt or small milk cartons or ice-cream. Only the much larger, established store chains like Keels Super provided users with an array of discounts or the Arpico store chain. Due to this, the most common and frequent type of discount is BOGO, and when looking at the geographical placement of big, established stores, they are often situated near main roads or populated areas.
- While most of the stores liked the idea of this service, two stores in particular opposed the idea.

Doing this survey revealed that there should be a large emphasis on the practical side of the solutions. Due to this, the algorithms could be more optimized and the conflicts between standardizing the APIs to work with different streams of data without the use of intermediate adapters or data translators seems to be infeasible. Following is a comparison between the considered data gathering methods.

Table 5.10 highlights the trade-offs, with API integrations being the most effective for real-time data, while crowd-sourcing and web scraping offer broader but less reliable coverage.

Table 5.10: Comparison of Data Collection Methods for Inventory Updates

Method	Description	Advantages	Challenges
API Integrations	Partner with stores for real-time inventory data via APIs.	Accurate, real-time, reliable.	Requires store cooperation, complex setup.
Crowd-Sourcing	Users report prices and availability, often gamified.	Broad coverage, cost-effective.	Less reliable, depends on user engagement.
Web Scraping	Collect data from store websites automatically.	Wide data source, initial coverage.	Legal issues, technical challenges.
Manual Updates	Stores update inventory via portal, less common for small stores.	Simple for stores, direct control.	Less reliable, time-consuming for stores.

6 Discussion and Conclusion

This section reflects on the overall project, interpreting the results in the context of the research objectives and the broader field, while also addressing limitations.

6.1 Discussion

When it comes to the modern day online shopping experience, most customers are numb to the idea of keeping up with the stores to finding the best deals available. That is because the current way that stores market these deals is through methods like mailing lists, ads and so on. These are bombarded on to customers and have lost their effect on most people except the diligent deal hunters. The innovations that have come about in the space that have succeeded the most are delivery services and online marketplaces that are centralized to their own stores or their value add only. The community benefits from these but there are also downsides as it demoralizes smaller shops and reduce the opportunity for innovation in the shopping experience. Searching for the items one wants at the store that there is a limited set of choices to choose from is the norm in online shopping. The advent of the smartphone has created a culture where customers are willing to share their location with service providers for convenience but the over promotion of delivery has created a gap in the market for the ones who would actually like to go to the store themselves. The in-store experience is devalued and the brick and mortar roadside shops are finding it difficult to adapt

to the new age. Groceries are where the project found that there is a great gap and the greatest amount of users that could be catered to. However the solution presented could also be adapted to other fields if it succeeds.

So the project proposes the solution as an application that takes what a customer needs to buy, checks the local stores that may have them in the customer's vicinity and shows the route that the customer should take to get the best deal in a convenient route. Additionally using the application for an extended period of time would allow the customer to just pick the items that the system will understand are what they usually buy. However this creates the need for the stores to be on-board as well. The system would need data and lots of it to serve the customers their best deals. Surveying stores showed that they are interested, especially if the addition of the system could help them adjust to the new age of e-commerce. Sri Lanka is still a mainly pen and paper based nation after all. However, they are not in unison about how exactly the data should be presented to the system. This sparked talk about standards. Standards that have shown success elsewhere and may benefit the local population as well. Say that the system actually has the data, then that data would need to be filtered effectively such that the customers are not overwhelmed. If the filtered data is irrelevant to the customer, the solution is a failure.

Taking these into consideration, as explained throughout the document, a system was developed. The system can take input from customers about what they want to buy and from stores about what they sell. Then after exploring the options there are to solve complex problems as mentioned above, algorithms as discussed earlier were chosen. A user friendly interface was designed. An architecture that could handle the potential load of a great amount of people all using the system was created.

The evaluation of the store recommendation and route planning system demonstrates its effectiveness in addressing the project's objectives. The parallel branch and bound algorithm, as detailed in Section 4.7, achieved near-optimal store recommendations with an average execution time of 0.096 seconds (Table 5.7), meeting the requirement for timely responses. This efficiency is crucial for real-time applications, ensuring users receive recommendations quickly. Furthermore, the route planning component, which combines an adaptive genetic algorithm with A* search, reduced travel distances by up to 25% (Figure 5.1), directly contributing to cost-effective shopping experiences.

These results align with and extend the existing literature. The hybrid approach to route planning builds on the foundational work of Hart, Nilsson, and Raphael 1968 and Tang et al. 2021 by incorporating real-time traffic data, addressing a significant limitation in static route planning methods. Similarly, the use of parallel branch and bound for store recommendations offers a practical alternative to exact methods like Integer Linear Programming (ILP), which can be computationally prohibitive for large-scale problems.

The implications of these findings are substantial. By providing users with optimized store suggestions and efficient routes, the system has the potential to reduce shopping time and costs, addressing the issues of information overload and price discrepancies highlighted in the motivation (Section 1.1). Moreover, the modular architecture (Section 3.1) ensures scalability, a feature lacking in existing chain-specific solutions such as those of Target or Walmart (Section 2.5).

However, the system is not without limitations. The reliance on API-based data collection, as discussed in Section 4.13, introduces challenges related to data accuracy and timeliness, as vendors may not update their inventories in real-time. Additionally, user adoption may be hindered by ingrained shopping habits, as revealed in the qualitative survey (Section 5.4). These challenges underscore the need for future work to refine data collection methods, possibly through IoT or POS integration, and to explore strategies for encouraging user engagement.

Looking ahead, several avenues for enhancement present themselves. Integrating public transport options or dynamic re-routing could further optimize the route planning component. Additionally, incorporating contextual factors such as store queues or real-time inventory changes could improve the system’s responsiveness to real-world conditions.

6.2 Conclusion

This research successfully developed and validated a system that enhances the grocery shopping experience through optimized store recommendations and efficient route planning. The key findings include the achievement of near-optimal store suggestions within 0.096 seconds (Table 5.7) and a reduction in travel distances by up to 25% (Figure 5.1). The SVD-based model for predicting future purchases further demonstrated

the system's personalization capabilities, achieving an F1-score of 0.85 (Figure 5.14). The primary contributions of this work lie in its integration of recommendation systems, route planning, and data standardization within a scalable, user-centric framework. By addressing the limitations of existing solutions and proposing a flexible approach to data collection, this research offers a practical solution tailored to the Sri Lankan retail context, with potential for global adaptation.

While challenges remain, particularly in data accuracy and user adoption, this project represents a significant step toward streamlining the grocery shopping experience. One such challenge is that stores have different data, and most stores have never taken the steps to make their stock appear online. As mentioned earlier, to cater to the stores that already have their own websites, the stores that use a point of sale system and the stores that barely have any computers at all, the system and the thinking behind it needs to be flexible. Due to this, room was left to expand the options for data ingestion in the future while having the basics set up in the scope of the project.

Another challenge is the fact that routing is its own beast when it comes to solving the problem at large. Simply using euclidean distance is useless when the customers need to travel in actual roads. So the use of available real world routing techniques were explored and integrated into the system. Creating a standard for all stores to adhere to when sharing data is a difficult challenge as well. Especially in the Sri Lankan climate where the good results of digital standardization has not become apparent to the populace yet. Another challenge is the fact that peoples' needs can change at any moment. If the system recommends you a route but on the way you remember something else to buy as well, how should the system be ready to handle this? If the stores report that they have something but someone comes in and buys all the stock before the customer gets there, how should the system handle that? The last few challenges are definitely the hardest to try to solve. But these are cases that do not disprove that the system can be effective in spite of them.

There are practical doubts about the system being effective in the real world. As the survey and simple observations of our surroundings reveal, people are creatures of habit. Going to the same stores that you usually go to and buying at whatever prices they offer is what people are used to and it would be greatly optimistic to think that that behavior will change with the introduction of this solution.

When it comes to the lessons that were learned during the process of developing the solution, the main one was that even the most mundane interaction in our daily lives can be exponentially more complex if enough focus were given to doing it as effectively as possible. At first the thought of choosing the best stores to buy groceries from seems extremely simple but doing it at scale while not wasting people's time is extremely difficult. The fact that users' voices being the most important thing to get a hold of at the start of a venture like this is another lesson that was learned. It would have prevented a lot of doubts of the effectiveness of the system had the survey been conducted earlier in the development cycle. Another lesson learned is that creating a new technique or even applying a rarely used technique can create unforeseen blocks in development. The project ran into such issues when the languages chosen to implement certain features did not have specific libraries or functionalities desired by the solution. And another great lesson was that scope must be kept solid as the mind wanders. There were many a time when refocusing on the core ideas were required to keep the project moving smoothly.

In conclusion the solution proposed and implemented to solve the problem of effective routing recommendation for in person shopping is a great first step towards a very complicated problem. And there is a great amount of work to be done to make the system be as polished as can be. And as was discussed, there is still more room for innovation in the online shopping space, even when the giants of the world seem to be dominating. Future refinements, such as enhanced data integration and contextual awareness, could further elevate its impact, underscoring the ongoing potential for innovation in this domain are discussed in detail under the next Section 7.

7 Future Directions

The project holds significant potential and can be extended into various other domains. However, the current system faces limitations, particularly in the methods of data collection and aggregation. Timely acquisition of store-product data or the implementation of recurring updates (periodic updates) is essential. To address this, the incorporation of intelligent web scrapers and IoT or embedded devices capable of transmitting data over long ranges is proposed. An initial concept involved establishing a peer-to-peer network, allowing clients to join ad hoc and retrieve necessary data from stores within the network, thereby identifying the most cost-efficient solution.

In the solver algorithms, explained in the earlier section, the OSRM server is being used to get the real-world, accurate travelling data. In the current system, in the filtering stages, a query is used to find the locations that are closest to the user using a built-in PostGIS function that considers the direct distance between the two points. This is not very accurate. Because in the real world, travelling distances are determined by the road networks. Even though a point may seem close to the user, it could be very expensive in terms of travelling time and cost to reach the store. In the current implementation, these cases are not being filtered in the initial filtering stages. Due to this, the algorithms perform additional calculations for unnecessary or incorrect solutions. Trajectory planning is also important. Assume the user was heading towards home, and the solver service is recommending solutions in the opposite direction, then in that case, it is not convenient for the user. The current system is not very concerned about such cases.

In the current system, the recommendation system is being used to predict a possible list of items that the user might purchase in the future. The next step is to incorporate the recommender service or a similar model to consider user preferences in an efficient manner. Due to this limitation in the system, even though the system is capable of predicting the items that a user might buy in the future, using this to better serve the user in the recommendation process and integrating the existing model or a new model is left as future work. Adding contextual understanding to the recommendation system is considered a future work, and the recommendation service will be aware of the user, and the recommendation service could leverage this data to provide more personalised predictions.

As mentioned previously mentioned under Section 4, the branch and bound algorithms can be distributed among many nodes. This could lead to increased performance and is considered future work in this context. As discussed before, making sure these algorithms work to give practical answers is more important. To achieve this, more survey data should be collected and studied to understand the patterns between using purchase patterns. Taking the real-world context into provide recommendations is very important for client retention. But there are two sides to this. In some cases, some of the constraints must be tightened, and some of the constraints or heuristics used by the algorithms should be more loosened. This could lead to over-approximation and under-approximation problems, thus it could lead to the suboptimal solution being presented to the users. And the algorithm could give biased answers. For example, just because customers like to travel less, recommending stores that provide the most items would leave more cost-efficient solutions. More research and survey data should be collected to find the right balance between the optimisation parameters.

In order to increase the overall quality of the recommendations and to avoid any disputes with the store owners, for example, think of a scenario like this: the application is recommending to the user some item with the sales value of \$100, which was entered to the system by the store owner, once the customer reach the store, the store owner may barter with the customer and ask for a much higher price, in such cases the store own has violated the terms of the system as well as the trust of the customer towards the application to gain a competitive edge by adding products at much lower prices, as a mitigation method, the system can use reviews and review scores in the recommendation process. The current system neither provides any review analysis method nor a review submission method. This is also considered as a future work.

Currently, the discount processing engine is not integrated into the main flow of the algorithm pipeline. Refining the discount processing engine and integrating the discount engine into the main flow could lead to solutions with added cost reductions. Even though this parameter is not directly used as it can be given as a secondary key for identifying a solution that has the highest possibility of having discounts for the items the user wants to buy.

Data extraction still remains an issue, as a mitigation methodology, the use of intelligent web scrapers or crawlers can be identified. Sometimes vendors may upload images

with product information, and crawlers have the ability to process such information could lead to higher amounts of information extraction and processing. All the information in a single method could help with the standardisation issues as well. The accumulation of the same information from different sources and processing it into a single format could lead to information with higher accuracy. Creating the necessary documents to streamline the standards, such as API documents, rule guidelines, and conducting developer workshops, is encouraged to ensure the audience captures the reasons behind standardisation as well as the motivations. This is a technique to avoid being over-reliant on vendors for data. Due to the unification of the standards, no matter what option the provider uses, the server will always receive a single format, using this, the system can make an informed decision whether to update the existing record (for example, inventory data) or not. As an indirect consequence of standardisation, this will serve as a multi-mode confirmation method for data validity. Integrating of inventory tracking systems discussed in the previous sections would also be a possible automation method.

Appendices

A Pseudocodes for Store Recommendation Algorithms

A.1 Naive Algorithm

```
FUNCTION generate_store_recommendation_naive(shopping_list,
    index)
    // Get item IDs from shopping list
    requested_item_ids = shopping_list.get('item_id')

    // Track which stores have which requested items
    store_inventory_map = empty dictionary

    // Process each item in inventory
    FOR EACH inventory_item IN items
        current_item_id = inventory_item[1]['item_id']
        current_quantity = inventory_item[1]['item_quantity']

        // Check if item is in our shopping list
        IF current_item_id IN requested_item_ids THEN
            current_store_id = inventory_item[1]['storeID']

            IF current_store_id IN store_inventory_map THEN
                store_inventory_map[current_store_id]
                    [current_item_id] = current_quantity
            ELSE
                store_inventory_map[current_store_id] =
                    {current_item_id: current_quantity}
            END IF
        END IF
    END FOR

    // Initialize result categories
    stores_with_all_items = empty array
```

```

stores_with_some_items = empty array
stores_with_no_items = empty array
total_item_count = 0
max_item_count = 0

// Categorize stores
FOR EACH current_store_id, store_items IN store_inventory_map
    matching_items = intersection(keys(store_items),
        requested_item_ids)

    IF size(matching_items) == size(requested_item_ids) THEN
        // Store has all items
        append [current_store_id, store_items, matching_items] to
            stores_with_all_items
    ELSE IF size(matching_items) > 0 THEN
        // Store has some items
        append [current_store_id, store_items, matching_items] to
            stores_with_some_items
    ELSE
        // Store has no items
        append [current_store_id] to stores_with_no_items
    END IF

    total_item_count = total_item_count + size(matching_items)
    max_item_count = maximum(max_item_count, size(matching_items))
END FOR

// Calculate average items per store
average_item_count = 0
IF size(store_inventory_map) > 0 THEN
    average_item_count = total_item_count / size(
        store_inventory_map)
END IF

```

```

// Return results
RETURN {
    stores_with_all_items,
    stores_with_some_items,
    stores_with_no_items,
    average_item_count,
    max_item_count
}
END FUNCTION

FUNCTION check_combinations(stores, shopping_list, min_results
    =2)
    viable_combinations = empty array

    // Try combinations of 2 to min(10, number of stores) stores
    FOR combination_size = 2 TO minimum(10, size(stores))

        FOR EACH store_combination IN combinations(stores,
            combination_size)
            remaining_items = set of shopping_list.get('item_id')
            item_store_mapping = empty string

            // Check if combination covers all items
            FOR EACH candidate_store IN store_combination
                FOR EACH item IN candidate_store[2]
                    IF item IN remaining_items THEN
                        item_store_mapping = item_store_mapping + item + ":"
                            + candidate_store[0] + "\n"
                        remove item from remaining_items
                    END IF
                END FOR
            END FOR
        END FOR
    END FOR

```

```

        // If all items are covered
        IF remaining_items is empty THEN
            append [store[0] for each store in store_combination] to
                viable_combinations

            IF size(viable_combinations) >= min_results THEN
                break inner loop
            END IF
        END IF
    END FOR

    IF size(viable_combinations) >= min_results THEN
        break outer loop
    END IF
END FOR

RETURN viable_combinations
END FUNCTION

{
    stores_with_all_items ,
    stores_with_some_items ,
    stores_with_no_items ,
    average_item_count ,
    max_item_count
} generate_store_recommendation_naive(shopping_list, index)

output stores_with_all_items + check_combinations(
    stores_with_some_items ,
    shopping_list ,
    min_results=2
)

```

A.2 Memory-Optimized Algorithm

```
FUNCTION to_string(array)
    // Converts an array of values to a string by mapping each
    value
    to its character representation
    RETURN concatenate(map each element in array to its character
        representation)
END FUNCTION

// Global variable to cache calculation results
calculated = [False, None]

FUNCTION pre_calculate_combinations()
    // Check if combinations were already calculated
    IF calculated[0] THEN
        RETURN calculated[1]
    END IF

    // Group items by store ID
    grouped_items = group items by 'storeID'

    // Dictionary mapping unique item combinations to store IDs
    store_item_combination_strings = empty dictionary

    // For each store, create a string representation of its
    unique items
    FOR EACH store_id, store_items IN grouped_items
        // Create a sorted string of item IDs this store carries
        item_set_string = to_string(sort(unique values of
            store_items['item_id']))

        // Add the store to the appropriate item combination entry
        IF item_set_string NOT IN store_item_combination_strings
            THEN
```



```

        store_item_combination_strings[item_set_string] = [
            store_id]
    ELSE
        append store_id to store_item_combination_strings[
            item_set_string]
    END IF
END FOR

// Cache the results
calculated = [True, store_item_combination_strings]
RETURN store_item_combination_strings
END FUNCTION

FUNCTION generate_store_recommendation_memory(shopping_list,
    index)
// Get pre-calculated store-item combinations
store_item_combination_strings = pre_calculate_combinations()

// Convert shopping list to a sorted string representation
item_ids = sort(unique values of shopping_list.get('item_id'))
item_string = to_string(item_ids)

// Initialize result containers
stores_with_matching_items = empty dictionary // hasSome
stores_with_all_items = empty array // hasAll
stores_with_no_items = empty array // hasNone
max_item_count = 0
total_item_count = 0

// Check each store's inventory against the shopping list
FOR EACH store_item_combination_string, store_ids IN
    store_item_combination_strings
        matching_item_count = 0
        matching_items_string = empty string

```

```

// Count matching items by character comparison
FOR EACH character IN item_string
    IF character IN store_item_combination_string THEN
        matching_item_count = matching_item_count + 1
        matching_items_string = matching_items_string +
            character
    END IF
END FOR

// Categorize the store based on item matches
IF matching_item_count == length(item_string) THEN
    // Store has all items
    append all store_ids to stores_with_all_items
ELSE IF matching_item_count > 0 THEN
    // Store has some items
    IF matching_items_string NOT IN stores_with_matching_items
        THEN
            stores_with_matching_items[matching_items_string] =
                store_ids
        ELSE
            append all store_ids to
                stores_with_matching_items[matching_items_string]
        END IF
    ELSE
        // Store has no items
        append all store_ids to stores_with_no_items
    END IF

// Update statistics
total_item_count = total_item_count + matching_item_count
max_item_count = maximum(max_item_count, matching_item_count
    )
END FOR

```

```

// Remove duplicates from arrays
stores_with_all_items = remove_duplicates(
    stores_with_all_items)
stores_with_no_items = remove_duplicates(stores_with_no_items)

// Collect all stores that have some items
stores_with_some_items = union of all
    store arrays in stores_with_matching_items
stores_with_some_items = remove_duplicates(
    stores_with_some_items)

// Find store combinations that cover all items
viable_combinations = greedy_check_combinations(
    stores_with_matching_items, item_string
)

// Return results
RETURN {
    stores_with_all_items,
    stores_with_matching_items,
    stores_with_no_items,
    total_item_count,
    max_item_count,
    viable_combinations
}
END FUNCTION

FUNCTION non_empty_substrings(string)
// Generates all non-empty substrings of a given string
substrings = empty array

FOR start = 0 TO length(string) - 1
    FOR end = start + 1 TO length(string)

```

```

        append string[start:end] to substrings
    END FOR
END FOR

RETURN substrings
END FUNCTION

FUNCTION greedy_check_combinations(stores_with_matching_items,
    item_string)
// Set of unique characters (items) in the shopping list
unique_items = set of characters in item_string

// Dictionary mapping item subsets to stores that have them
store_item_power_map = empty dictionary
previous_key_count = 0

// Build power set of item combinations and associated stores
FOR iteration = 1 TO 10
    FOR item_subset, store_ids IN stores_with_matching_items
        // Generate all substrings of the item subset
        FOR substring IN non_empty_substrings(item_subset)
            IF substring IN store_item_power_map THEN
                append all store_ids to store_item_power_map[substring
                    ]
            ELSE
                store_item_power_map[substring] = store_ids
            END IF
        END FOR
    END FOR
END FOR

// Remove duplicate store IDs
FOR EACH key, value IN store_item_power_map
    store_item_power_map[key] = remove_duplicates(value)
END FOR

```

```

// Check if we've added any new substrings
current_key_count = size(store_item_power_map)
IF current_key_count == previous_key_count THEN
    output "iteration " + iteration
    break loop
END IF
previous_key_count = current_key_count
END FOR

// Convert store arrays to sets for faster operations
FOR EACH key, value IN store_item_power_map
    store_item_power_map[key] = convert value to set
END FOR

// Look for complementary store combinations
viable_combinations = empty array

max_substring_length = length(item_string) / 2

// Sort keys by length in descending order
FOR item_subset IN sort store_item_power_map keys by length (
    descending)
    subset_items = set of characters in item_subset

    // Find the complement of items not in this subset
    missing_items = unique_items - subset_items
    missing_items_string = sort and join missing_items into
        string

    // Check if we have stores that cover the missing items
    IF missing_items_string IN store_item_power_map THEN
        append (
            store_item_power_map[item_subset],

```

```

        store_item_power_map[missing_items_string]
    ) to viable_combinations
END IF

// Optimization: stop checking once substrings are too short
IF length(item_subset) <= max_substring_length THEN
    break loop
END IF
END FOR

Return viable_combinations

END FUNCTION

{
    stores_with_all_items ,
    stores_with_matching_items ,
    stores_with_no_items ,
    total_item_count ,
    max_item_count ,
    viable_combinations
} = generate_store_recommendation_memory(shopping_list , index)

output stores_with_all_items + viable_combinations

```

A.3 Filtered Greedy Algorithm

```
FUNCTION generate_recommendations(shopping_list)
  // Extract item IDs from shopping list
  requested_item_ids = get item IDs from shopping_list

  // Get items data from all stores
  store_inventory = get_store_items_data()

  // Filter out items not in the shopping list
  store_inventory = keep only rows where item_id is in
    requested_item_ids

  // Process each item in shopping list to adjust quantities and
    prices
  FOR EACH item IN shopping_list
    requested_quantity = get quantity of item
    current_item_id = get item_id of item

    // Remove stores that don't have enough quantity
    store_inventory = keep only rows where
      (item_quantity >= requested_quantity OR item_id !=
        current_item_id)

    // Multiply price by requested quantity
    FOR EACH row IN store_inventory where item_id equals
      current_item_id
      row.item_price = row.item_price * requested_quantity
    END FOR
  END FOR

  // Group items by store_id and calculate total prices
  store_summaries = empty list
  FOR EACH store_id IN unique store_ids from store_inventory
```

```

    items_in_store = get all rows from store_inventory with this
        store_id
    item_set = get unique item_ids from items_in_store
    total_price = sum of item_price from items_in_store
    item_count = count of items in item_set

    add {store_id, item_set, total_price, item_count} to
        store_summaries
END FOR

// Sort by number of items
sort store_summaries by item_count in descending order

// Find stores that have all requested items
stores_with_all_items = empty list
stores_with_some_items = empty list

FOR EACH store IN store_summaries
    IF store.item_count equals count of items in shopping_list
        THEN
            add store to stores_with_all_items
        ELSE
            add store to stores_with_some_items
        END IF
END FOR

// Sort stores with all items by price
sort stores_with_all_items by total_price in ascending order

// Get set of all requested item IDs for easier comparison
requested_items_set = set of requested_item_ids

// Initialize recommendations structure

```



```

recommendations = create dictionary with empty "combinations"
list

// For each possible item count (from max to min)
FOR k = count of items in shopping_list - 1 DOWN TO 2
    // Check stores that have at most k items
    FOR EACH store IN stores_with_some_items where item_count <=
        k
        // Find missing items for this store
        missing_items = items in requested_items_set
                        that are not in store.item_set

        // Find stores that have the missing items
        found_missing_stores = empty list

        // Check each missing item
        FOR EACH item IN missing_items
            // Find stores carrying this item
            stores_with_item = get rows from store_inventory
                                where item_id equals
                                item

            // If no store has this item, can't complete the
            shopping list
            IF stores_with_item is empty THEN
                break inner loop
            END IF

            // Add the first store that has this item
            add first store from stores_with_item to
                found_missing_stores
        END FOR

    // If we found stores for all missing items

```

```

IF count of found_missing_stores equals count of
    missing_items THEN
    // Track additional stores needed and their costs
    additional_stores = empty dictionary

    // Process each store with missing items
    FOR i = 0 TO count of found_missing_stores - 1
        store_id = get store_id from found_missing_stores[i]
        item_price = get item_price from found_missing_stores[
            i]
        item_id = get item_id from found_missing_stores[i]

        IF store_id exists in additional_stores THEN
            // Update existing store entry
            additional_stores[store_id].price += item_price
            add item_id to additional_stores[store_id].items
        ELSE
            // Create new store entry
            additional_stores[store_id] = {
                price: item_price,
                items: [item_id]
            }
        END IF
    END FOR

    // Add this combination to recommendations
    add {
        main_store_id: store.store_id,
        main_item_ids: store.item_set,
        main_item_price_total: store.total_price,
        extra_stores: additional_stores
    } to recommendations.combinations
END IF

```

```
        break inner loop
    END FOR
END FOR

// Add stores with all items to recommendations
recommendations.hasAll = stores_with_all_items

RETURN recommendations
END FUNCTION

INPUT shopping_list

output generate_recommendations(shopping_list).recommendations
```

A.4 Branch and Bound Algorithm

Input: UserInput $U = (I, q, l_u, r, m_u, N)$, Stores S

Output: List of top N Recommendations

1. Initialize:

```
- item_ids = [i.id for i in I]
- item_quantities = {i.id: q_i for i in I}
- dist_matrix = precompute_distances(l_u, S)
- Bits = bitset(item_ids)
- store_items = {
    s.id: Bits([
        i for i in item_ids
        if s.stock[i] >= q_i and m_u in s.payment_methods
    ])
    for s in S
}
- all_items = Bits(item_ids)
- item_to_stores = {
    i: sort([
        (s, s.prices[i]) for s in S
        if s.stock[i] >= q_i and m_u in s.payment_methods
    ], key=price)
    for i in I
}
- min_item_costs = {
    i: item_to_stores[i][0][1] * q_i for i in I
}
- best_score = infinity
- candidates = []
- store_set = {}
```

2. Function `branch_and_bound(store_ids, covered, assignment, total_cost)`:

```
- uncovered = all_items \ covered
```

```

- est_cost = sum(
    min_item_costs[i] for i in uncovered
    if min_item_costs[i] >= 0
)
- if any(min_item_costs[i] < 0 for i in uncovered): return
- tour_dist = calc_tour_distance(store_ids, l_u, dist_matrix)
- lower_bound = total_cost + tour_dist + est_cost
- if lower_bound >= best_score: return
- if covered == all_items:
    - store_tuple = tuple(sorted(store_ids))
    - if store_tuple not in store_set:
        - store_set.add(store_tuple)
        - rec_stores = [s in S where s.id in store_ids]
        - candidates.append(
            Recommendation(
                rec_stores, assignment,
                total_cost, tour_dist
            )
        )
        - best_score = min(best_score, total_cost + tour_dist
    )
    - return
- for each store s in S:
    - if s.id not in store_ids and m_u in s.payment_methods:
        - new_store_ids = store_ids + (s.id,)
        - new_covered = covered \cup store_items[s.id]
        - new_assignment = copy(assignment)
        - new_cost = total_cost
        - for each item_id in new_covered \ covered:
            - valid_stores = [
                (s_id, price) for s_id in new_store_ids
                if item_id in store_items[s_id]
            ]
            - if valid_stores:

```

```

        - (best_store_id, best_price) = min(
            valid_stores, key=price
        )
        - new_assignment[item_id] = store with id
            best_store_id
        - new_cost += best_price * q[item_id]
    - branch_and_bound(
        new_store_ids,
        new_covered,
        new_assignment,
        new_cost
    )

```

3. Call `branch_and_bound((), Bits([]), {}, 0.0)`
4. Return `sorted(candidates, key=score)[:N]`

A.5 Beam Search Algorithm

Input: UserInput $U = (I, q, l_u, r, m_u, N)$, Stores S

Output: List of top N Recommendations

1. Initialize:

```
- item_ids = [i.id for i in I]
- item_quantities = {i.id: q_i for i in I}
- dist_matrix = precompute_distances(l_u, S)
- Bits = bitset(item_ids)
- store_items = {
    s.id: Bits([
        i for i in item_ids
        if s.stock[i] >= q_i and m_u in s.payment_methods
    ])
    for s in S
}
- all_items = Bits(item_ids)
- item_to_stores = {
    i: sort([
        (s, s.prices[i]) for s in S
        if s.stock[i] >= q_i and m_u in s.payment_methods
    ], key=price)
    for i in I
}
- min_item_costs = {
    i: item_to_stores[i][0][1] * q_i for i in I
}
- BEAM_WIDTH = 100
- candidates = []
- store_set = {}
- beam = [((), Bits([]), {}, 0.0)]
```

2. While beam is not empty:

```
- new_beam = []
```

```

- For each (store_ids, covered, assignment, total_cost) in
  beam:
    - If covered == all_items:
      - store_tuple = tuple(sorted(store_ids))
      - If store_tuple not in store_set:
        - store_set.add(store_tuple)
        - rec_stores = [s in S where s.id in store_ids]
        - tour_dist = calc_tour_distance(
            store_ids, l_u, dist_matrix
          )
        - candidates.append(
            Recommendation(
              rec_stores, assignment,
              total_cost, tour_dist
            )
          )
      - Continue
    - For each store s in S:
      - If s.id not in store_ids and m_u in s.
        payment_methods:
          - new_store_ids = store_ids + (s.id,)
          - new_covered = covered \cup store_items[s.id]
          - new_assignment = copy(assignment)
          - new_cost = total_cost
          - For each item_id in new_covered \ covered:
            - valid_stores = [
                (s_id, price) for s_id in new_store_ids
                if item_id in store_items[s_id]
              ]
            - If valid_stores:
              - (best_store_id, best_price) = min(
                  valid_stores, key=price
                )
              - new_assignment[item_id] = store

```



```

with id
best_store_id

- new_cost += best_price * q[item_id]
- uncovered = all_items \ new_covered
- est_cost = sum(
    min_item_costs[i] for i in uncovered
    if min_item_costs[i] >= 0
)
- If any(min_item_costs[i] < 0 for i in uncovered
):
    continue
- tour_dist = calc_tour_distance(
    new_store_ids, l_u, dist_matrix
)
- lower_bound = new_cost + tour_dist + est_cost
- new_beam.append((
    new_store_ids, new_covered,
    new_assignment, new_cost
))

- Sort new_beam by (total_cost + tour_distance)
- beam = new_beam[:BEAM_WIDTH]

3. Return sorted(candidates, key=score)[:N]

```

A.6 Integer Linear Programming Algorithm

The following pseudocode outlines the Integer Linear Programming (ILP) logic used to solve the multi-store recommendation problem. It optimizes for cost while adhering to constraints such as item availability and store selection.

```
Input: UserInput U = (I, q, l_u, r, m_u, N), Stores S, Redis
      client
Output: List of top N Recommendations

1. Initialize:
   - item_ids = [i.id for i in I]
   - item_quantities = {i.id: q_i for i in I}
   - dist_matrix = precompute_distances(l_u, S, redis_client)
   - model = Gurobi.Model("MultiStoreRecommendation")
   - store_select = {s.id: binary_var for s in S}
   - item_assign = {(i, s.id): binary_var for i in I for s in S
                     if s.stock[i] >= q_i and m_u in s.payment_methods}
   - candidates = []
   - store_set = {}

2. Set Objective:
   - cost = sum(item_assign[i, s.id] * s.prices[i] * q_i for i
                 in I for s in S if (i, s.id) in item_assign)
   - model.setObjective(cost, GRB.MINIMIZE)

3. Add Constraints:
   - For each i in I:
       sum(item_assign[i, s.id] for s in S if (i, s.id) in
           item_assign) == 1
   - For each (i, s.id) in item_assign:
       item_assign[i, s.id] <= store_select[s.id]
   - sum(store_select[s.id] for s in S) >= 1

4. For n = 1 to N:
   - model.optimize() with timeout
```

```

- If solution exists:
    selected_stores = [s for s in S if store_select[s.id] >
        0.5]
    store_ids = tuple(sorted([s.id for s in selected_stores])
        )
    If store_ids not in store_set:
        assignment = {}
        total_cost = 0.0
        For each i in I:
            For s in selected_stores:
                If item_assign[i, s.id] > 0.5:
                    assignment[i] = s
                    total_cost += (s.prices[i] * q_i)
            tour_dist = calc_tour_distance(store_ids, l_u,
                dist_matrix)
            candidates.append(Recommendation(selected_stores,
                assignment, total_cost, tour_dist))
            store_set.add(store_ids)
        Add constraint:
            sum(store_select[s.id] for s in store_ids) <= len(
                store_ids) - 1
- Else: break
- If timeout: break

```

```

5. Return sorted(candidates, key=score)[:N]

```

A.6.1 Attempted Implementation

An attempted was carried out to fully implement and integrate the Integer Linear Programming methods to solve the multi-store recommendation problem. Due to the nature of the problem, resulting code could not handle large datasets and was considered as infeasible. This algorithm is a part of a matrix-based method to solution to solve multi-store recommendation problems, by modeling the problem in to a matrix based format. Initial algorithm can be found below.

```
from ortools.linear_solver import pywraplp
import random

def solve_store_item_problem(costs):
    """
    args:
        costs (list of list of float): A 2D array where
            costs[i][j] is the cost of buying item j from
            store i.
    returns:
        list of tuples: Optimal assignments in the form
            (store, item, cost).
    """
    num_stores = len(costs)
    num_items = len(costs[0])

    solver = pywraplp.Solver.CreateSolver("SCIP")
    if not solver:
        raise Exception("Solver not available!")

    # Create decision variables: x[i][j] indicates whether
    # item j is bought from store i
    x = []
    for i in range(num_stores):
        x.append([
            solver.IntVar(0, 1, f"x[{i}][{j}]")
```

```

        for j in range(num_items)
    ])

    # Objective: Minimize total cost
    objective = solver.Objective()
    for i in range(num_stores):
        for j in range(num_items):
            objective.SetCoefficient(x[i][j], costs[i][j])
    objective.SetMinimization()

    for j in range(num_items):
        solver.Add(
            solver.Sum(
                x[i][j] for i in range(num_stores)
            ) == 1
        )

    status = solver.Solve()

    # results checks
    if status == pywraplp.Solver.OPTIMAL:
        solution = []
        total_cost = 0
        for i in range(num_stores):
            for j in range(num_items):
                if x[i][j].solution_value() > 0:
                    solution.append((i, j, costs[i][j]))
                    total_cost += costs[i][j]
        return solution, total_cost
    else:
        raise Exception("No optimal solution found!")

if __name__ == "__main__":
    # Example costs: 3 stores and 3 items

```

```

costs = [
    [10, 20, 15], # Store 1
    [12, 18, 25], # Store 2
    [14, 16, 22], # Store 3
]

solution, total_cost = solve_store_item_problem(costs)
print("Optimal assignments:")
for store, item, cost in solution:
    print(
        f"Store {store + 1}: Buy Item {item + 1} "
        f"(Cost: {cost})"
    )
print(f"Total Cost: {total_cost}")

num_items = 1000
num_stores = 1000

cost_matrix = [
    [
        random.randint(700, 2000)
        for _ in range(num_items)
    ]
    for _ in range(num_stores)
]

solution, total_cost = solve_store_item_problem(cost_matrix)
print("Optimal assignments:")
for store, item, cost in solution:
    print(
        f"Store {store + 1}: Buy Item {item + 1} "
        f"(Cost: {cost})"
    )
print(f"Total Cost: {total_cost}")

```

B Psuedocodes for Route Planning Algorithms

B.1 A* Algorithm

```
function get_distance(loc1, loc2, store_id1, store_id2):  
    # Create cache key from location coordinates  
    key = (  
        loc1.latitude,  
        loc1.longitude,  
        loc2.latitude,  
        loc2.longitude  
    )  
  
    # Return cached result if available  
    if key in distance_cache:  
        return distance_cache[key]  
  
    if google_maps_client exists:  
        # Get distance and duration from Google Maps API  
        result = google_maps_client.get_distance_matrix(loc1,  
            loc2)  
        distance = result.distance  
        duration = result.duration_in_traffic  
    else:  
        # Calculate Haversine distance  
        distance = haversine_distance(loc1, loc2)  
        duration = distance / average_speed # Convert to time  
  
    # Apply traffic factor  
    traffic_factor = traffic_data[store_id1 or store_id2] or 1.0  
    duration = duration * traffic_factor  
  
    # Cache and return result  
    distance_cache[key] = (distance, duration)  
    return distance, duration
```

```

function heuristic(current_store_id, remaining_store_ids):
    if no remaining stores:
        return 0

    current_store = find store by current_store_id
    if not current_store:
        return infinity

    min_distance = infinity
    for store_id in remaining_store_ids:
        store = find store by store_id
        distance = haversine_distance(
            current_store.location,
            store.location
        )
        min_distance = min(min_distance, distance)

    return min_distance

function find_path(store_order):
    if no store_order:
        return empty path, 0 distance, 0 time

    current_location = user_location
    total_distance = 0
    total_time = 0
    path = empty list

    for store_id in store_order:
        store = find store by store_id
        if not store:
            throw error "Store not found"

```



```
        distance, time = get_distance(
            current_location,
            store.location,
            store_id
        )
        total_distance += distance
        total_time += time
        path.append(store_id)
        current_location = store.location

# Return to starting point
distance, time = get_distance(current_location,
    user_location)
total_distance += distance
total_time += time

return path, total_distance, total_time
```

B.2 Genetic Algorithm

```
function generate_individual():
    # Select random combination
    combination_idx = random_index(combinations)
    store_ids = list of store IDs from combination
    shuffle(store_ids)
    return (combination_idx, store_ids)

function fitness(individual):
    combination_idx, store_order = individual
    astar.stores = combinations[combination_idx].stores
    path, distance, time = astar.find_path(store_order)
    return distance, time, path

function tournament_selection(population, tournament_size):
    # Select random individuals for tournament
    tournament = random_sample(population, tournament_size)
    best_individual = null
    best_distance = infinity

    for individual in tournament:
        distance, _, _ = fitness(individual)
        if distance < best_distance:
            best_distance = distance
            best_individual = individual

    return best_individual

function crossover(parent1, parent2):
    if random() > crossover_rate:
        return parent1, parent2

    # Choose combination from one parent
    child_idx = random_choice(
```

```

        parent1.combination_idx,
        parent2.combination_idx
    )
    store_ids = list of store IDs from combinations[child_idx]

    if store_ids length <= 1:
        return (child_idx, store_ids), (child_idx, store_ids)

    # Perform order crossover
    start, end = sorted random indices in store_ids
    child_order = empty list of store_ids length
    child_order[start:end] = store_ids[start:end]

    # Fill remaining positions
    remaining = store_ids not in child_order
    for store_id in remaining:
        find next empty position in child_order
        place store_id in position

    return (child_idx, child_order), (child_idx, child_order)

function mutate(individual):
    combination_idx, store_order = individual

    # Mutate combination
    if random() < mutation_rate and multiple combinations exist:
        combination_idx = random_index(combinations)
        store_order = list of store IDs from new combination
        shuffle(store_order)

    # Mutate order
    if random() < mutation_rate and multiple stores:
        swap two random positions in store_order

```

```

    return (combination_idx, store_order)

function run():
    # Initialize population
    population = generate population_size individuals
    best_individual = null
    best_distance = infinity
    fitness_history = empty list
    diversity_history = empty list

    for each generation:
        new_population = empty list
        fitnesses = empty list

        # Evaluate fitness
        for individual in population:
            distance, time, path = fitness(individual)
            fitnesses.append(distance)
            if distance < best_distance:
                best_distance = distance
                best_individual = (
                    individual.combination_idx,
                    path,
                    time
                )

        # Track statistics
        fitness_history.append(average(fitnesses))
        diversity = unique store orders / population_size
        diversity_history.append(diversity)

        # Select elites
        elites = top elite_size individuals by fitness
        new_population.extend(elites)

```

```

    # Generate new population
    while new_population not full:
        parent1 = tournament_selection(population)
        parent2 = tournament_selection(population)
        child1, child2 = crossover(parent1, parent2)
        child1 = mutate(child1)
        child2 = mutate(child2)
        new_population.append(child1, child2)

    population = new_population trimmed to population_size

# Final validation
    best_idx, best_path, best_time = best_individual
    distance, time, path = fitness((best_idx, best_path))

    statistics = {
        fitness_history,
        best_fitness_history,
        diversity_history,
        generations_executed,
        final_diversity,
        convergence_generation
    }

    return best_idx, best_path, time, distance, statistics

```

C Discount Engine

C.1 CFG of Rule Language(DCDQL)

```
RULE := CONDITION [LOGICAL_OPERATOR CONDITION] THEN
      DISCOUNT_TYPE VALUE
CONDITION := KEY COMPARISON_OPERATOR VALUE
KEY := min_cart_price | total_price | product_id | KEY_D
KEY_D := total_category_price | category_id | purchase_quantity
COMPARISON_OPERATOR := = | != | > | < | >= | <= | IN
VALUE := NUMBER | STRING | [LIST]
LIST := NUMBER | STRING
LOGICAL_OPERATOR := AND | OR
DISCOUNT_TYPE := product_percentage | product_flat_amount | DT_T
DT_T := cart_percentage | cart_flat_amount | BOGO
NUMBER := Integer | Float
STRING := [a-zA-Z][a-zA-Z0-9_]*
```

C.2 Discount Engine

```
// Discount Rule Engine
// Handles rules like: "product_id IN [1,2,3] THEN
    product_percentage 10"

// Data Structures
STRUCT Token
    type: STRING    // e.g., KEYWORD, NUMBER, LBRACKET
    value: STRING    // e.g., "product_id", "1"

STRUCT Condition
    key: STRING      // e.g., "product_id"
    operator: STRING // e.g., "=", "IN"
    value: ANY        // NUMBER, STRING, LIST

STRUCT Rule
    conditions: LIST of Condition // Max 2
    logical_op: STRING             // "AND", "OR", ""
    discount_type: STRING          // e.g., "product_percentage"
    discount_value: NUMBER         // e.g., 10

// Main Function
FUNCTION ProcessDiscountRule(
    rule_string,
    product,
    cart
) RETURNS Discount
    tokens = Lex(rule_string)
    rule = Parse(tokens)
    discount = Evaluate(rule, product, cart)
    RETURN discount

// Lexer: Tokenizes input string
```

```

FUNCTION Lex(rule_string) RETURNS LIST of Token
    tokens = EMPTY LIST
    current = EMPTY STRING
    in_list = FALSE

    FOR each char c in rule_string
        IF c is WHITESPACE
            IF current is NOT EMPTY
                tokens.ADD(CreateToken(current))
                current = EMPTY
            END IF
        ELSE IF c is "["
            in_list = TRUE
            tokens.ADD(Token(type="LBRACKET", value="["))
            current = EMPTY
        ELSE IF c is "]"
            in_list = FALSE
            IF current is NOT EMPTY
                tokens.ADD(CreateToken(current))
            END IF
            tokens.ADD(Token(type="RBRACKET", value="]"))
            current = EMPTY
        ELSE IF c is "," and in_list
            IF current is NOT EMPTY
                tokens.ADD(CreateToken(current))
                current = EMPTY
            END IF
            tokens.ADD(Token(type="COMMA", value=","))
        ELSE
            current = current + c
        END IF
    END FOR

    IF current is NOT EMPTY

```



```

        tokens.ADD(CreateToken(current))
    END IF

    RETURN tokens

FUNCTION CreateToken(value) RETURNS Token
    keywords = ["min_cart_price", "product_id",
                "category_id", "total_price",
                "total_category_price",
                "purchase_quantity"]
    ops = ["=", "!", ">", "<", ">=", "<=", "IN"]
    logical = ["AND", "OR"]
    discounts = ["product_percentage",
                 "product_flat_amount",
                 "cart_percentage",
                 "cart_flat_amount", "BOGO"]

    IF value in keywords
        RETURN Token(type="KEYWORD", value=value)
    ELSE IF value in ops
        RETURN Token(type="OPERATOR", value=value)
    ELSE IF value in logical
        RETURN Token(type="LOGICAL_OP", value=value)
    ELSE IF value = "THEN"
        RETURN Token(type="THEN", value=value)
    ELSE IF value in discounts
        RETURN Token(type="DISCOUNT_TYPE", value=value)
    ELSE IF value MATCHES NUMBER
        RETURN Token(type="NUMBER", value=value)
    ELSE IF value MATCHES STRING
        RETURN Token(type="STRING", value=value)
    ELSE
        ERROR("Invalid token: " + value)
    END IF

```

```

// Parser: Builds rule from tokens
FUNCTION Parse(tokens) RETURNS Rule
    rule = NEW Rule
    index = 0

    condition1 = ParseCondition(tokens, index)
    rule.conditions.ADD(condition1)
    index = condition1.next_index

    IF index < tokens.LENGTH and tokens[index].type = "
        LOGICAL_OP"
        rule.logical_op = tokens[index].value
        index = index + 1
        condition2 = ParseCondition(tokens, index)
        rule.conditions.ADD(condition2)
        index = condition2.next_index
    END IF

    IF index >= tokens.LENGTH or tokens[index].type != "THEN"
        ERROR("Expected THEN")
    END IF
    index = index + 1

    IF index >= tokens.LENGTH or tokens[index].type != "
        DISCOUNT_TYPE"
        ERROR("Expected discount type")
    END IF
    rule.discount_type = tokens[index].value
    index = index + 1

    IF index >= tokens.LENGTH or tokens[index].type != "NUMBER"
        ERROR("Expected discount value")
    END IF

```

```

rule.discount_value = tokens[index].value
index = index + 1

IF index < tokens.LENGTH
    ERROR("Unexpected tokens")
END IF

RETURN rule

FUNCTION ParseCondition(tokens, start_index) RETURNS Condition
    condition = NEW Condition
    index = start_index

    IF index >= tokens.LENGTH or tokens[index].type != "KEYWORD"
        ERROR("Expected keyword")
    END IF
    condition.key = tokens[index].value
    index = index + 1

    IF index >= tokens.LENGTH or tokens[index].type != "OPERATOR"
        "
        ERROR("Expected operator")
    END IF
    condition.operator = tokens[index].value
    index = index + 1

    IF condition.operator = "IN"
        condition.value = ParseList(tokens, index)
        index = condition.value.next_index
    ELSE
        IF index >= tokens.LENGTH or
            tokens[index].type not in ["NUMBER", "STRING"]
            ERROR("Expected number or string")
        END IF
    END IF

```

```

        condition.value = tokens[index].value
        index = index + 1
    END IF

    RETURN Condition(key=condition.key,
                    operator=condition.operator,
                    value=condition.value,
                    next_index=index)

FUNCTION ParseList(tokens, start_index) RETURNS LIST
    list = EMPTY LIST
    index = start_index

    IF index >= tokens.LENGTH or tokens[index].type != "LBRACKET
    "
        ERROR("Expected [")
    END IF
    index = index + 1

    WHILE index < tokens.LENGTH and tokens[index].type != "
    RBRACKET"
        IF tokens[index].type in ["NUMBER", "STRING"]
            list.ADD(tokens[index].value)
            index = index + 1
        ELSE
            ERROR("Expected number or string")
        END IF
        IF index < tokens.LENGTH and tokens[index].type = "COMMA
        "
            index = index + 1
        END IF
    END WHILE
END WHILE

```

```

IF index >= tokens.LENGTH or tokens[index].type != "RBRACKET
"
    ERROR("Expected "]")
END IF
index = index + 1

RETURN LIST(value=list, next_index=index)

// Evaluator: Applies rule to product and cart
FUNCTION Evaluate(rule, product, cart) RETURNS Discount
    result = TRUE

    FOR each condition in rule.conditions
        value = GetContextValue(condition.key, product, cart)
        cond_result = EvaluateCondition(condition, value)
        IF rule.logical_op = "AND"
            result = result AND cond_result
        ELSE IF rule.logical_op = "OR"
            result = result OR cond_result
        ELSE
            result = cond_result
        END IF
    END FOR

    IF result
        RETURN ComputeDiscount(rule.discount_type,
                                rule.discount_value,
                                product, cart)
    ELSE
        RETURN NO_DISCOUNT
    END IF

FUNCTION GetContextValue(key, product, cart) RETURNS ANY
    CASE key

```

```

    WHEN "min_cart_price"
        RETURN cart.total_price
    WHEN "total_price"
        RETURN product.price
    WHEN "product_id"
        RETURN product.id
    WHEN "total_category_price"
        RETURN cart.category_totals[product.category]
    WHEN "category_id"
        RETURN product.category
    WHEN "purchase_quantity"
        RETURN product.quantity
    ELSE
        ERROR("Unknown key: " + key)
END CASE

```

```

FUNCTION EvaluateCondition(condition, value) RETURNS BOOLEAN

```

```

    CASE condition.operator
        WHEN "="
            RETURN value = condition.value
        WHEN "!="
            RETURN value != condition.value
        WHEN ">"
            RETURN value > condition.value
        WHEN "<"
            RETURN value < condition.value
        WHEN ">="
            RETURN value >= condition.value
        WHEN "<="
            RETURN value <= condition.value
        WHEN "IN"
            RETURN value IN condition.value
        ELSE
            ERROR("Unknown operator")
    END CASE

```

```

END CASE

FUNCTION ComputeDiscount(type, value, product, cart) RETURNS
Discount
CASE type
    WHEN "product_percentage"
        amount = product.price * value / 100
        RETURN Discount(type="product", amount=amount)
    WHEN "product_flat_amount"
        RETURN Discount(type="product", amount=value)
    WHEN "cart_percentage"
        amount = cart.total_price * value / 100
        RETURN Discount(type="cart", amount=amount)
    WHEN "cart_flat_amount"
        RETURN Discount(type="cart", amount=value)
    WHEN "BOGO"
        IF product.quantity >= 2
            RETURN Discount(type="product",
                            amount=product.price)
        ELSE
            RETURN NO_DISCOUNT
        END IF
    ELSE
        ERROR("Unknown discount type")
END CASE

```

D Psuedocodes of Data Generation Scripts

D.1 Route Planning Algorithm Dataset Generation

```
Initialize logging system to track events and info

DEFINE FUNCTION generate_store(id, lat, lon, items):
    RETURN a dictionary representing a store with:
        - unique ID
        - location (latitude & longitude)
        - random distance value
        - a set of items

DEFINE FUNCTION generate_dataset(name, num_combinations,
    stores_per_combination):
    Log start of dataset generation with name and parameters

    INITIALIZE dataset dictionary with:
        - empty recommendation data
        - empty traffic indices
        - fixed user location
        - genetic algorithm parameters
        - Google Maps flag set to false

    IF dataset name is "edge_empty":
        Leave recommendation data empty
        Log that empty dataset is generated
    ELSE:
        Initialize store_id_counter to 1
        FOR each combination in range(num_combinations):
            Initialize an empty list of stores
            FOR each store in range(stores_per_combination):
                Generate random latitude and longitude within
                    bounds
                Construct store ID
```



```

        Create a random number (1-3) of items with:
            - IDs, units, prices, quantities, and costs
        Create a store using generate_store()
        Add the store to the list of stores
        Assign a random traffic index for the store
        Increment store_id_counter
    Add this list of stores to the recommendation data

    Log the number of stores generated per combination

Ensure the output directory exists
Define output file path based on dataset name
Save the dataset as a JSON file to the output path
Log that dataset was saved

VERIFY:
    - File exists
    - File is not empty
    IF any checks fail:
        Log error and raise appropriate exception

TRY:
    Call generate_dataset() with:
        - "small": 4 combinations, 5 stores each
        - "medium": 10 combinations, 12 stores each
        - "large": 20 combinations, 20 stores each
        - "edge_single_store": 1 combination, 1 store
        - "edge_empty": 0 combinations, 0 stores
    CATCH exceptions and log any errors during generation
    RAISE the exception again

```

References

- Aditya, PH, Indra Budi, and Qorib Munajat (2016). “A comparative analysis of memory-based and model-based collaborative filtering on the implementation of recommender system for E-commerce in Indonesia: A case study PT X”. In: *2016 International Conference on Advanced Computer Science and Information Systems (ICACSIS)*. IEEE, pp. 303–308.
- Anandhan, Anitha et al. (2018). “Social media recommender systems: review and open research issues”. In: *IEEE Access* 6, pp. 15608–15628.
- Baluch, Anna (Aug. 2022). *Retail Inventory Management Best Practices*. en-US. Section: Business. URL: <https://www.forbes.com/advisor/business/retail-inventory-management/> (visited on 04/26/2025).
- Beam search* (Oct. 2024). en. Page Version ID: 1248868876. URL: https://en.wikipedia.org/w/index.php?title=Beam_search&oldid=1248868876 (visited on 04/26/2025).
- Branch and bound* (Apr. 2025). en. Page Version ID: 1284698095. URL: https://en.wikipedia.org/w/index.php?title=Branch_and_bound&oldid=1284698095 (visited on 04/26/2025).
- Burke, Robin (2000). “Knowledge-based recommender systems”. In: *Encyclopedia of library and information systems* 69.Supplement 32, pp. 175–186.
- (2002). “Hybrid recommender systems: Survey and experiments”. In: *User modeling and user-adapted interaction* 12, pp. 331–370.
- Çelik, Özer (Sept. 2018). “A Research on Machine Learning Methods and Its Applications”. In: DOI: [10.31681/jetol.457046](https://doi.org/10.31681/jetol.457046).
- Chulyadyo, Rajani and Philippe Leray (2014). “A personalized recommender system from probabilistic relational model and users’ preferences”. In: *Procedia Computer Science* 35, pp. 1063–1072.

- Data Exchange Mechanisms and Considerations* (2020). en. URL: <https://enterprisearchitecture.harvard.edu/data-exchange-mechanisms>.
- De Pessemier, Toon et al. (2015). “Combining collaborative filtering and search engine into hybrid news recommendations”. In: *3rd International Workshop on News Recommendation and Analytics (INRA 2015), in conjunction with the 9th ACM Conference on Recommender Systems (RecSys 2015)*, pp. 13–18.
- Do, Minh-Phung Thi, DV Nguyen, and Loc Nguyen (2010). “Model-based approach for collaborative filtering”. In: *6th International conference on information technology for education*, pp. 217–228.
- Gal, Michal S and Daniel L Rubinfeld (2018). “DATA STANDARDIZATION”. en. In.
- General Transit Feed Specification* (2025). URL: <https://gtfs.org/> (visited on 04/26/2025).
- Global Trade Item Number* (2025a). en. Page Version ID: 1278604263. URL: https://en.wikipedia.org/w/index.php?title=Global_Trade_Item_Number&oldid=1278604263.
- Global Trade Item Number* (Mar. 2025b). en. Page Version ID: 1278604263. URL: https://en.wikipedia.org/w/index.php?title=Global_Trade_Item_Number&oldid=1278604263 (visited on 04/26/2025).
- GS1* (Jan. 2025). en. URL: <https://en.wikipedia.org/w/index.php?title=GS1&oldid=1267367519> (visited on 04/26/2025).
- Gupta, Rajiv (Feb. 2023). “Research Paper on Artificial Intelligence”. In: *International Journal of Engineering and Computer Science* 12, pp. 25654–20656. DOI: [10.18535/ijecs/v12i02.4720](https://doi.org/10.18535/ijecs/v12i02.4720).
- Hameed, Mohd Abdul, Omar Al Jadaan, and Sirandas Ramachandram (2012). “Collaborative filtering based recommendation system: A sur-

- vey”. In: *International Journal on Computer Science and Engineering* 4.5, p. 859.
- Hart, Peter E., Nils J. Nilsson, and Bertram Raphael (1968). “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”. In: *IEEE Transactions on Systems Science and Cybernetics* 4.2, pp. 100–107. DOI: [10.1109/TSSC.1968.300136](https://doi.org/10.1109/TSSC.1968.300136).
- Huttner, Joseph (2009). “From Tapestry to SVD: A Survey of the Algorithms That Power Recommender Systems”. PhD thesis.
- Integer programming* (Apr. 2025). en. Page Version ID: 1285601109. URL: https://en.wikipedia.org/w/index.php?title=Integer_programming&oldid=1285601109 (visited on 04/26/2025).
- Isinkaye, F.O., Y.O. Folajimi, and B.A. Ojokoh (2015). “Recommendation systems: Principles, methods and evaluation”. In: *Egyptian Informatics Journal* 16.3, pp. 261–273. ISSN: 1110-8665. DOI: <https://doi.org/10.1016/j.eij.2015.06.005>. URL: <https://www.sciencedirect.com/science/article/pii/S1110866515000341>.
- Kaelbling, Leslie Pack, Michael L Littman, and Andrew W Moore (1996). “Reinforcement learning: A survey”. In: *Journal of artificial intelligence research* 4, pp. 237–285.
- Kardan, Ahmad A. and Maryam Hooman (2013). “Targeted advertisement in social networks using recommender systems”. In: *7th International Conference on e-Commerce in Developing Countries:with focus on e-Security*, pp. 1–13. DOI: [10.1109/ECDC.2013.6556728](https://doi.org/10.1109/ECDC.2013.6556728).
- Khanal, Shristi Shakya et al. (2020). “A systematic review: machine learning based recommendation systems for e-learning”. In: *Education and Information Technologies* 25, pp. 2635–2664.

- Kong, Xiangjie et al. (2018). “Shared Subway Shuttle Bus Route Planning Based on Transport Data Analytics”. In: *IEEE Transactions on Automation Science and Engineering* 15.4, pp. 1507–1520. DOI: [10.1109/TASE.2018.2865494](https://doi.org/10.1109/TASE.2018.2865494).
- Kunaver, Matevž and Tomaž Požrl (2017). “Diversity in recommender systems—A survey”. In: *Knowledge-based systems* 123, pp. 154–162.
- Langley, Pat, Wayne Iba, Kevin Thompson, et al. (1992). “An analysis of Bayesian classifiers”. In: *Aaii*. Vol. 90. Citeseer, pp. 223–228.
- Lightspeed (2023). *Lightspeed Retail: POS and Commerce Solutions*. URL: <https://www.lightspeedhq.com/> (visited on 04/26/2025).
- Linear programming (Feb. 2025). en. Page Version ID: 1278106560. URL: https://en.wikipedia.org/w/index.php?title=Linear_programming&oldid=1278106560 (visited on 04/26/2025).
- Liu, Qiong and Ying Wu (Jan. 2012). “Supervised Learning”. In: DOI: [10.1007/978-1-4419-1428-6_451](https://doi.org/10.1007/978-1-4419-1428-6_451).
- Locally (2025). en. URL: <https://www.locally.com> (visited on 04/26/2025).
- Murphy, Lauren et al. (Oct. 2018). “API Designers in the Field: Design Practices and Challenges for Creating Usable APIs”. en. In: *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Lisbon: IEEE, pp. 249–258. ISBN: 978-1-5386-4235-1. DOI: [10.1109/VLHCC.2018.8506523](https://doi.org/10.1109/VLHCC.2018.8506523). URL: <https://ieeexplore.ieee.org/document/8506523/>.
- Naeem, Samreen et al. (Apr. 2023). “An Unsupervised Machine Learning Algorithms: Comprehensive Review”. In: *IJCDS Journal* 13, pp. 911–921. DOI: [10.12785/ijcds/130172](https://doi.org/10.12785/ijcds/130172).
- NewStore - A Unified Commerce Platform for Global Brands (2025). en-US. URL: <https://www.newstore.com/> (visited on 04/26/2025).

- NIEM Open For Government / An OASIS Open Project* (2025). URL: <https://www.niem.gov/> (visited on 04/26/2025).
- Open Food Facts* (2025). en. URL: <https://github.com/openfoodfacts> (visited on 04/27/2025).
- OpenStreetMap on GitHub* (2025). en. URL: <https://github.com/openstreetmap> (visited on 04/27/2025).
- Papadakis, Harris et al. (2022). “Collaborative filtering recommender systems taxonomy”. In: *Knowledge and Information Systems* 64.1, pp. 35–74.
- Pazzani, Michael J and Daniel Billsus (2007). “Content-based recommendation systems”. In: *The adaptive web: methods and strategies of web personalization*. Springer, pp. 325–341.
- Protocol Buffers* (2025). en. URL: <https://protobuf.dev/> (visited on 04/26/2025).
- Rich, Elaine (1979). “User modeling via stereotypes”. In: *Cognitive science* 3.4, pp. 329–354.
- Safavi, Sadaf, Mehrdad Jalali, and Mahboobeh Houshmand (2022). “Toward Point-of-Interest Recommendation Systems: A Critical Review on Deep-Learning Approaches”. In: *Electronics* 11.13. ISSN: 2079-9292. DOI: [10.3390/electronics11131998](https://doi.org/10.3390/electronics11131998). URL: <https://www.mdpi.com/2079-9292/11/13/1998>.
- Salunke, Tanmayee and Unnati Nichite (Dec. 2022). “Recommender Systems in E-commerce”. In: DOI: [10.13140/RG.2.2.10194.43202](https://doi.org/10.13140/RG.2.2.10194.43202).
- Schafer, Ben, Joseph Konstan, and John Riedl (Oct. 1999). “Recommender Systems in E-Commerce”. In: *1st ACM Conference on Electronic Commerce, Denver, Colorado, United States*. DOI: [10.1145/336992.337035](https://doi.org/10.1145/336992.337035).

- Singh, Pradeep et al. (Jan. 2021). “Recommender Systems: An Overview, Research Trends, and Future Directions”. In: *International Journal of Business and Systems Research* 15, pp. 14–52. DOI: [10.1504/IJBSR.2021.10033303](https://doi.org/10.1504/IJBSR.2021.10033303).
- Srinivas, M. and Lalit M. Patnaik (1994). “Adaptive Probabilities of Crossover and Mutation in Genetic Algorithms”. In: *IEEE Transactions on Systems, Man, and Cybernetics* 24.4, pp. 656–667. DOI: [10.1109/21.286385](https://doi.org/10.1109/21.286385).
- ST, Red Rocket and Perishable (Feb. 2025). *How RFID is Revolutionizing Retail: Loss Prevention & Inventory Accuracy*. en-US. URL: <https://datascan.com/the-benefits-of-real-time-inventory-tracking-for-grocery-stores/> (visited on 04/26/2025).
- Steck, Harald et al. (Sept. 2021). “Deep Learning for Recommender Systems: A Netflix Case Study”. In: *AI Magazine* 42, pp. 7–18. DOI: [10.1609/aimag.v42i3.18140](https://doi.org/10.1609/aimag.v42i3.18140).
- Storemapper: Customizable Store Locator App and Software* (2025). en. URL: <https://www.storemapper.com/> (visited on 04/26/2025).
- Tang, Gang et al. (Mar. 2021). “Geometric A-Star Algorithm: An Improved A-Star Algorithm for AGV Path Planning in a Port Environment”. In: *IEEE Access* PP, pp. 1–1. DOI: [10.1109/ACCESS.2021.3070054](https://doi.org/10.1109/ACCESS.2021.3070054).
- Target (2025). *Target Store Locator*. URL: <https://www.target.com/store-locator/find-stores> (visited on 06/24/2025).
- Thorat, Poonam B, Rajeshwari M Goudar, and Sunita Barve (2015). “Survey on collaborative filtering, content-based filtering and hybrid recommendation system”. In: *International Journal of Computer Applications* 110.4, pp. 31–36.

- Tong, Yongxin et al. (July 2018). “A unified approach to route planning for shared mobility”. In: *Proc. VLDB Endow.* 11.11, pp. 1633–1646. ISSN: 2150-8097. DOI: [10.14778/3236187.3236211](https://doi.org/10.14778/3236187.3236211). URL: <https://doi.org/10.14778/3236187.3236211>.
- Verbert, Katrien et al. (2012). “Context-aware recommender systems for learning: a survey and future challenges”. In: *IEEE transactions on learning technologies* 5.4, pp. 318–335.
- Walmart (2025). *Walmart Store Finder*. URL: <https://www.walmart.com/store-finder> (visited on 06/24/2025).
- Yang, Wan-Shiou, Hung-Chi Cheng, and Jia-Ben Dia (2008). “A location-aware recommender system for mobile shopping environments”. In: *Expert Systems with Applications* 34.1, pp. 437–445. ISSN: 0957-4174. DOI: <https://doi.org/10.1016/j.eswa.2006.09.033>. URL: <https://www.sciencedirect.com/science/article/pii/S0957417406002934>.