Computational Framework for Solving the Meal Delivery Routing Problem

Sebastián Quintero Rojas (s.quintero425@uniandes.edu.co)

Department of Industrial Engineering
Universidad de Los Andes. Bogotá, Colombia

2020-12-13

Abstract

The Meal Delivery Routing Problem (MDRP) is a problem in which an online restaurant aggregator receives orders from diners and matches couriers that perform the pick-up and drop-off of these requests. These operations have become more popular over the past few years and on-demand delivery has gained special traction during the COVID-19 pandemic. There are many challenges involved in this problem: the order arrival stream is highly dynamic and uncertain, the fleet works under the gig economy model in which they have the freedom to reject requests and log on and off as they please, most orders are expected to be delivered in under 40 minutes and there are stakeholders with conflictive interests. In this research a computational framework is presented to handle an environment where solutions to the MDRP may be tested. At the core, there is a discrete events simulator which accurately represents the components of a meal delivery operation. The simulator has blocks where policies are embedded, that represent how actors make decisions or take actions. The proposed framework is modular, hence specific blocks may be interchanged so that different policies can be compared or new ones introduced. The computational framework is designed to transparently load instances and inputs, execute the simulation and output performance metrics. In addition, the MDRP is given new definitions. Lastly, real-life instances are provided for testing.

Keywords: On-Demand Delivery, Meal Delivery, Dynamic Vehicle Routing, Matching, Discrete Events Simulation.

1 Introduction

On-demand delivery is a logistic problem in which customers order goods and services via mobile or web applications that connect them to providers. There is a system controller which receives the requests and assigns them to facilitators that visit the service providers to execute a pick-up and arrive at the customer's location to finalize the order with the corresponding drop-off. In this context, meal delivery is one of the most important on-demand delivery industries, where orders are placed by customers at a volatile and uncertain rate. This problem is framed in the gig economy model, where freelance couriers enable the delivery of these meals.

Mobile applications are projected to generate approximately 935 billion USD in revenue in 2023 (Kumar, 2020). Moreover, the global food delivery mobile app market is expected to hit 16.6 billion USD by 2023, at a compound-annual growth rate of 27.9 percent (Klein, 2019). In the recent context of the COVID-19 pandemic, US sales for meal delivery services grew 135 percent in October of 2020 (Yeo, 2020). In smaller markets such as Colombia, the COVID-19 pandemic has shown an increase of up to 300 percent in e-commerce platforms (Forbes, 2020). Big players such as Uber Eats, GrubHub and Delivery Hero showcase the same challenges as smaller companies such as Rappi, Grab or Glovo: delivering meals as fast as possible, with a high level of service quality by orchestrating a fleet of

freelance couriers. These statistics speak not only about the market opportunity of doing research on the meal delivery business but also its impact on customers.

The problem of delivering meals to customers has been addressed in several works (Reyes et al., 2018; Steever et al., 2019; Ulmer et al., 2017; Yildiz & Savelsbergh, 2019), although not as extensively as more traditional problems (e.g., VRP). The Meal Delivery Routing Problem (MDRP) captures several aspects of the operation in meal delivery services. There are four stakeholders: the customers ordering meals, the restaurants that prepare them, the couriers who deliver the food and the company, or aggregator, who coordinates the other stakeholders. Each of them hold a different goal they pursue and behave accordingly, leading to multiple challenges. Customers want their meals delivered on-time, maintaining the quality of the food. Restaurants want a courier placed in their location to guarantee a cooked meal will be delivered without delay, as they have a continuous stream of orders coming in. Independent couriers want to achieve high earnings for delivering orders and are picky about which requests they accept, adding uncertainty in the operation. Lastly, the company who aggregates these requests must orchestrate every aspect of the operation, without having any downtime and making decisions under stochastic conditions. In short, meal delivery presents a highly complex transportation problem where a large number of variables interact among each other.

Top delivery companies have arrival rates for orders that range from tens to hundreds per minute, deriving in a challenge that is not only analytical (developing prescriptive and predictive models) but also technical (having a scalable and robust software architecture). This leads to an important aspect which is understanding the system as a whole and the interactions that take place, before designing the solutions that attack the logistic problems mentioned. Part of this aspect consists of making available real-life data for further research. Some academic research relies on artificially creating instances that do not reflect real locations and times, but rather follow a probability distribution, such that outcomes obtained under these conditions are ideal at best and instance size tends to be very small. Given the volume of the operation, it is important to test solutions in a controlled environment that does not affect production. Devising strategies to prepare the system in advance can provide an edge over the usual myopic decisions that are made by businesses. It is challenging to establish a holistic framework where the delivery problem can be studied and treated, given that most solutions are tailor-made for a single context.

On the face of this challenge, this research proposes a computational framework that captures the dynamics of the MDRP, with a discrete events simulator at its core. The framework allows for input loading, solution processing and output saving, enabling robust and informed simulation of MDRP solutions. The framework should be applicable to real-life conditions, thus a set of real instances are provided by the company Rappi and constitute the main input of the framework. This is done to encourage the development of solutions from real, as opposed to purely academic, data. In this fashion, real locations and time stamps constitute the inputs of the instances and the complete volume of data reflects the dimensions of the problem. An important component of the framework is the World: a module that orchestrates the arrival of new customers and couriers and advances the internal clock of the system. The simulator embeds policies to simulate how decisions or actions are made. A policy is a set of rules, algorithms, models or any other type of analytical solution that govern an action. The simulator is designed to have interchangeable policies that fulfill different needs. Policies are conceived as building blocks in the framework: they can be plugged in or out and the definition of standard inputs and outputs facilitate their interaction with the World. With this, further research can focus on developing novel and interesting policies. The framework outputs a set of widely used performance metrics, with the aim of using them as a comparison tool to decide which policies provide the best solution for a specific system condition. The design makes the framework lightweight and highly versatile: it can represent a specific time window within a day, with the potential of simulating full weeks of operation. In addition to the framework, new definitions are given to the MDRP, with missing dynamics from a real-life operation. All of these contributions face the challenge that is understanding the computational complexity of the MDRP and providing a platform where hypotheses can be tested aiming at better understanding the variability in the processes and the impact of adopted decisions and policies.

The rest of this document is organized as follows:

- Section 2 contains a review of the related literature.
- Section 3 specifies the guidelines of this research.
- Section 4 formally describes the definition of the MDRP, including structural assumptions and performance metrics.
- Section 5 describes the framework along with the simulator and proposed policies.
- Section 6 describes the instances provided, how they are used in computational experiments and discusses the results of these experiments.
- Section 7 sums up the contents of this research while providing concluding remarks and future recommendations.
- Appendix A provides a glossary for terminology.
- Appendix B provides the configurations for test scenarios.
- Appendix C describes how to benchmark the computers used in this research.

2 Literature Review

This research focuses on the Meal Delivery Routing Problem, which is a class of Dynamic Vehicle Routing Problem (DVRP) that incorporates pickups and deliveries, belonging to the Dynamic Pickup and Delivery Problems (DPDP). Given the nature and urgency of the context of the problem, it is closely associated with the Same Day Delivery Problem (SDDP) and the Dial-A-Ride Problem (DARP), in which customers are transported, rather than goods. A survey over these problems is conducted to study the state of the art pertaining the MDRP.

2.1 Dynamic Vehicle Routing and Pickup & Delivery Problems

Pillac et al. (2013) provide a comprehensive survey of the DVRP. Problems are classified under the taxonomy of information evolution and quality, classifying problems into: static-deterministic, static-stochastic, dynamic-deterministic or dynamic-stochastic. In this line, Pillac et al. (2012) propose an event-driven framework to optimize the DVRP, applying multi-core and multi-threaded parallelism. The framework is used for a problem with stochastic information, where customer demands are random variables. Psaraftis et al. (2016) also compile a review on the literature of the DVRP and classify different publications by these problem features: type of problem (as pointed by Pillac et al. (2013)), logistical context (pickup & deliveries), transportation mode (road, maritime, air, etc…), objective function (min cost, min distance, min travel time, etc…), fleet size (single, limited number, large size), time constraints (hard, soft, unclear, no), vehicle capacity constraints (yes or no), ability to reject customers (yes or no), nature of dynamic element (requests, travel / service time, vehicle availability), nature of stochasticity (location, demand, time or no) and solution methods (Tabu search, neighborhoods search, insertion, dynamic programming, Markov decision process, other).

An earlier review of the DPDP was done by Berbeglia et al. (2010). Solution strategies and algorithmic performance assessment is discussed. An application for transporting handicapped people is presented but nowadays there exists a variety of applications, specially in the on-demand delivery spectrum. Mitrović-Minić and Laporte (2004) explore four waiting strategies for a DPDP with time windows: drive-first, wait-first, dynamic waiting and advanced dynamic waiting. These waiting strategies reference the scheduling of drivers when deciding at what moment they should depart for the next location. Yan et al. (2019) present a dynamic waiting model, in which dynamic pricing helps solve a ride-hailing problem. In this environment, data from the company Uber is studied, an operation governed by the DPDP.

2.2 Same Day Delivery and Dial A Ride Problems

The SDDP was introduced by Voccia et al. (2015). To solve the SDDP, a Markov decision process is presented. Future information is incorporated in the route planning to account for the highly dynamic nature of the problem. A key difference between the SDDP and DPDP is that in the SDDP, vehicles must always return to the depot (single pickup location). Related to the SDDP, Klapp et al. (2018) formulate the Dynamic Dispatch Waves Problem (DDWP). There is a depot where orders arrive dynamically and must be delivered on the same day. Decisions are made by waves (moments in time), where vehicles can be dispatched or held back. Information incorporated in the solution is deterministic.

In this context, Castillo et al. (2017) explain the Wild Goose Chase (WGC) phenomenon, in which drivers follow the demand of rides in a ride-hailing operation and during demand spikes, service quality is extremely affected. A surge-pricing model is introduced where the ride's price is dynamically adjusted based on the demand of a geographical zone. This has become industry standard for ride-hailing services. The problem of transporting on-demand passengers is known as the Dial-A-Ride Problem (DARP). A survey on DARP literature is done by Cordeau and Laporte (2007). Once again, a common example for transporting elderly or disabled people is mentioned, but with the rise of ride-hailing platforms, there are now a great variety of realizations of the DARP.

2.3 Meal Delivery Problems

The former problems lead to the current challenge of delivering meals by on-line services. Reyes et al. (2018) formally introduce the Meal Delivery Routing Problem and is the foundation of this research. They provide a myopic rolling horizon approach to solving the problem, first obtaining routes and then matching them to couriers. Instances are artificially created relying on euclidean distances and a simplified coordinate system from a reference point, with sizes ranging to the hundreds of orders. Ulmer et al. (2017) introduce the Restaurant Meal Delivery Problem (RMDP). The problem is very similar to the MDRP but varies in that for the RMDP, the time it takes to cook an order is stochastic. To solve the RMDP, a cost function approximation is proposed based on a route-based Markov decision process model.

Yildiz and Savelsbergh (2019) follow the MDRP and create an exact solution with a column-generation approach. However, this solution assumes a clairvoyant dispatcher with perfect information, which is an unlikely scenario to encounter in production. This solution can be used for shorter decision periods for scheduled orders, although this feature is not considered in the MDRP. Steever et al. (2019) define the Virtual Food Court Delivery Problem (VFCDP), in which multiple restaurants may be included in a single customer's order. An MIP formulation is provided to solve this problem in conjunction with an auction-based heuristic to handle proactive decisions that incorporate the future demand. Steever et al. (2019) show that incorporating look-ahead policies outperforms a purely myopic solution approach. Instances provided are artificially created based on probability distributions and correspond to a small operation.

Liu (2019) presents an MIP model for delivering meals with a fleet of drones, as opposed to using humans. This model allows for the advantage of avoiding traffic conditions and uncertainty in the behavior of couriers, whereas the performance of these drones is highly limited and charging cycles are part of important decision variables. Liao et al. (2020) introduce the Green Meal Delivery Routing Problem (GMDRP) where one of the objectives is to minimize the carbon footprint of the solution. It uses a genetic algorithm, principal component analysis and clustering to find initial delivery routing and further optimization is done with Adaptive Neighborhood Search. It is a class of Multi-Objective VRP. Hildebrandt and Ulmer (2020) dwell in the MDRP but focus on the aspect of predicting arrival times for customers. Both an offline and online approach is conducted over a computational study that shows how proper predictions improve the performance of the system.

3 Research Guidelines

After surveying the related literature in the context of the MDRP, there are multiple solutions focused on solving the routing problems with varying degrees of complexity. Understanding the problem is essential, and although all works have made a thorough effort in providing novel models and algorithms that solve the dynamic routing problem, these tend to focus on the model itself. Businesses do not thrive as a consequence of applying the best or most complex algorithm. On the contrary, companies succeed by understanding the operation and industry they belong to. This leads to the opportunity and challenge of developing a holistic testing environment where the problem can be framed and analyzed, as well as providing a means to evaluate solution strategies. Testing and measuring solutions is *as* important as their development, thus the following objective guides this research:

- Develop a computational framework where solutions for the Meal Delivery Routing Problem can be tested and gaps from the existing literature regarding operational dynamics are closed.

To achieve this objective, the following secondary objectives are stated:

1. Define operational dynamics missing from the existing MDRP literature.
2. Develop a discrete events simulator to represent the operation in which the MDRP functions.
3. Prepare real-life data instances to test solutions and feed the simulator.
4. Develop policies for matching orders to couriers.
5. Develop policies to capture the movement mechanics of couriers in the city.
6. Develop policies that model the acceptance or rejection of notifications by couriers.
7. Develop policies for modeling how orders are canceled by actors.
8. Design a computational framework where policies can be easily interchanged and operates with the simulator at its core.
9. Test different solution policies over a variety of scenarios using the proposed framework to obtain simulated performance metrics for policies comparison.

4 Problem Description

To understand how the computational framework is established, first the MDRP is described. In addition to the definition that is reviewed from the literature, new contributions are made to capture missing aspects from a real-life operation and the corresponding assumptions are given.

4.1 The Meal Delivery Routing Problem (MDRP)

The MDRP is considered as defined by Reyes et al. (2018) and restated here for completeness. It is paraphrased in a different manner to adapt it to this research, while keeping all of its elements and logic.

Let U be the set of users, who want meals delivered and place orders, with each user $u \in U$ having a drop-off location ℓ_u. Let R be the set of restaurants, where meals must be picked up, and each restaurant $r \in R$ has a pick-up location ℓ_r. Then, let O be the set of orders placed by the users. Each order $o \in O$ has an associated restaurant $r_o \in R$ and user $u_o \in U$, a placement time a_o, a ready time e_o, a pick-up service time s^r (depending on the restaurant) and a drop-off service time s^u (linked to the user). Let C be the set of couriers, who are used to deliver the orders in O. Let $c_o \in C$ be the courier assigned to order $o \in O$, noting that a courier may eventually *not* be assigned. Each courier $c \in C$ has an on-time e_c, an on-location ℓ_c, and an off-time l_c, with $l_c > e_c$. A courier's compensation at the end of the shift is defined as $max\{p_1 m_c, p_2(l_c - e_c)\}$, where p_1 is a fixed compensation per order delivered, m_c is the number of orders delivered during the shift and p_2 is a fixed compensation rate

per hour. Orders from the same restaurant may be aggregated into bundles, or routes, where each route has a single pick-up but multiple drop-off locations. Let S be the set of routes. Any route $s \in S$ must fulfill $|\{r_o, \forall o \in S\}| = 1$. All information about R and C is completely known. The information about a user and its associated order is only revealed at placement time a_o. In summation, the MDRP consists of:

> *"…determining feasible routes for couriers to complete the pick-up and delivery of orders, with the objective to optimize a single or multiple performance measures."*
>
> —(Reyes et al., 2018)

In the operating environment, the following two practices take place:

1. *Prepositioning.* A courier can be notified of an instruction to pick up & drop off a route or to move to a new location, changing the courier's current position.
2. *Assignment Updates.* When a courier is picking up an order at a restaurant, new notifications to pick up additional orders may be received, as long as they belong to the same restaurant.

4.2 Novel Characteristics

Contributing to the definition of the problem, the following novel features, which are not considered in the literature are added to accurately represent a real-life operation:

1. Couriers are able to freely move about the city when idling, actively making decisions of whether and where they want to move. At every time fraction f_r, this is $t \mod f_r = 0, \forall t \in T$, the courier evaluates if they want to relocate. Let $\ell_{c,t}$ be the location of courier $c \in C$ at time $t \in T$, then $\ell_{c,t+1} = \mathbb{P}_{me}(\ell_{c,t})$ is the location of courier $c \in C$ at time $t+1$ governed by the *movement evaluation* policy $\mathbb{P}_{me}$, whose output may well be $\ell_{c,t+1} = \ell_{c,t}$ or $\ell_{c,t+1} \neq \ell_{c,t}$.
2. The dispatcher can notify pick-up & drop-off or prepositioning requests. This feature is considered in the MDRP but lacks formal definition. Let N_t be the set of notifications at time $t \in T$. Each notification $n \in N_t$ can be of type prepositioning (n_p) or pick-up & drop-off (n_{pd}), thus $n \in \{n_p \cap n_{pd} = \emptyset\}$. Each notification has an associated location ℓ_n. Let c_n be the courier and s_n the route associated to notification $n \in N$. For prepositioning notifications, the location is the projected destination and for pick-up & drop-off notifications, the location is the first stop of the route, this is, the pick-up location ℓ_r.
3. Couriers may accept or reject notifications generated by the system. Let h_{ℓ_1,ℓ_2} be the haversine distance between location ℓ_1 and ℓ_2. Given that a courier may accept or reject a notification, each courier $c \in C$ has an acceptance rate p_c. A courier accepts a notification based on probability $P_c = f(h_{\ell_{c,t},\ell_n}, p_c)$, with $P_c > 0$. This acceptance function has a positive domain over $h_{\ell_{c,t},\ell_n}$ and p_c. The acceptance function fulfills that $P_c \propto 1/h_{\ell_{c,t},\ell_n}$ and $P_c \propto h_c$. One can see that a courier tends to reject a notification proportionately to having a lower acceptance rate or it being far away.
4. Each courier has a vehicle that alters the movement mechanics around the city. Let V be the set of vehicles and courier $c \in C$ has a vehicle $v_c \in V$.
5. Each order has a preparation time in which the meal *begins* cooking. The preparation time is denoted as d_o for order $o \in O$, and it fulfills $a_o \leq d_o$. The difference with the placement time a_o is that the system keeps the order from the restaurant r_o until d_o. This opens up the MDRP to the possibility of having scheduled orders (such as groceries), where $a_o \ll d_o$. In the context of the original MDRP, $a_o = d_o$.
6. A user $u \in U$ has a cancellation time t^u. After $d_o + t^u$, if the order $o \in O$ has not been assigned to a courier, the user u_o may decide to cancel it based on a user cancellation policy $\mathbb{P}_{uc}$ whose output is $\{0, 1\}$.

7. The system (or dispatcher as it is referred to in further sections) has a cancellation time t^d. The dispatcher is the agent in charge of controlling the operation and thus is responsible for maintaining appropriate service levels. After $d_o + t^d$, where $t^d > t^u$ and the order has no courier assigned, the dispatcher may decide to cancel the order based on the dispatcher's *cancellation* policy $\mathbb{P}_{dc}$ whose output is $\{0, 1\}$.
8. Every order $o \in O$ has an individual expected drop-off time f'_o, which is a different approach to having a generalized target drop-off time τ that would be the same for all orders. The actual drop-off time of the order is f_o. One goal of the solution of the MDRP must be to minimize the expected drop-off lateness for all orders, this is $|f'_o - f_o| \approx 0$.
9. The dispatcher evaluates sending pick-up & drop-off notifications at every time fraction f, this is $t \mod f = 0, \forall t \in T$. For a rolling horizon approach (Reyes et al., 2018), $f > 0$ and the time fraction can be minutes. For a clairvoyant approach (Yildiz & Savelsbergh, 2019), where all decisions are made at a single moment, $f = |T|$. In addition, for an events-based solution (Pillac et al., 2012), $f \approx 0$ and information is optimized as soon as it arrives. This definition of f allows flexibility in defining solutions for the MDRP.
10. The dispatcher evaluates sending prepositioning notifications at every time fraction f_p, $t \mod f_p = 0, \forall t \in T$, and similar configurations to the previously described f can be achieved.

4.3 Structural Assumptions

The following assumptions are considered in the definition of this modified version of the MDRP:

- Any route $s \in S$ has a limited number of orders S_{max}.
- The travel time b_{ℓ_1,ℓ_2} between any pair of locations ℓ_1 and ℓ_2 is invariant over time.
- A courier follows a city's real street layout while moving and respects traffic signals, independent of their vehicle.
- Couriers cannot be notified of new instructions after their off-time l_c has passed but can finish any ongoing assignments.
- Couriers execute autonomous decisions, as described in characteristics 1 and 3 of the **novel characteristics** section.
- Couriers can only be notified if they are idling or picking up an order (or group of orders). If a courier is moving, due to safety concerns, the real-life practice of not notifying them is applied, as well as when a courier is dropping off an order, so they can focus on the task at hand.
- A user can only order once throughout the day, this is, any user has a single order associated to them. In other problems, such as the Virtual Food Court Delivery Problem (Steever et al., 2019), users can have several orders simultaneously.
- A courier may only receive one notification at the same time.
- An order may only be assigned to a single courier.
- Prepositioning notifications and assignment updates are encouraged and performed by the dispatcher.
- Once an order is canceled, either by the user or the dispatcher, it cannot return to the system and negatively impacts service metrics.
- A courier's vehicle is invariant over time.
- The system's information regarding U, R, C and O is deterministic.

4.4 Performance Metrics

To measure a solution for the MDRP the following performance metrics are considered as written by Reyes et al. (2018):

1. Number of orders delivered.
2. Total courier compensation.

3. Cost per order: total courier compensation divided by number of orders delivered.
4. Fraction of couriers receiving guaranteed minimum compensation.
5. Click-to-door time: the difference between the drop-off time and the placement time of an order.
6. Click-to-door time overage: the difference between the drop-off time of an order and the placement time of an order plus the target click-to-door time.
7. Ready-to-door time: the difference between the drop-off time of an order and the ready time of an order.
8. Ready-to-pickup time: the difference between the pick-up time of an order and its ready time.
9. Courier utilization: the fraction of the courier duty time that is devoted to driving, pick-up service, and drop-off service (as opposed to time spent waiting).
10. Courier delivery earnings: courier earnings when considering only the number of orders served.
11. Courier compensation: the maximum of the guaranteed minimum compensation (based on the length of the duty period) and the delivery earnings.
12. Orders delivered per hour: for each courier, the number of orders delivered divided by the length of the shift.
13. Bundles picked up per hour: for each courier, the number of order bundles assigned divided by the length of the shift.
14. Orders per bundle: for each assignment, the number of orders to be picked up.

New performance metrics are added to capture important outcomes:

1. In-store-to-pickup time: the difference between the pick-up time and the time at which the courier arrives at the store. This metric shows how in time are the couriers arriving to the restaurants.
2. Drop-off lateness: the absolute difference between the order's expected drop-off time and the actual drop-off time. This metric captures quality of service for the user.
3. Number of canceled orders.
4. Fulfillment: delivered orders divided by the total orders (delivered and canceled). A fulfillment of 100% is the target for any company.
5. Click-to-taken-time: the difference between the time an order is accepted by a courier and the placement time. This metric captures how the global acceptance probability of the system behaves.

5 Methodology

5.1 Computational Framework

The computational framework to solve the MDRP is an integrated software solution that consists of three modules:

1. *World Mock*. This module of the framework is the interface that brings new data to the system. Real-life instances are loaded here and new data is obtained: users and couriers logging on to the system. The Internal clock advances time.
2. *Reality Simulation*. This module contains the discrete events simulator. All the subtleties of how different actors interact with one another happen here. This module saves the performance metrics of the solution. Policies for the simulator are loaded in this module.
3. *Services*. Services that provide functionality to the simulation. Services are self contained and are characterized by receiving an input and generating an output, without other dependencies.

The computational framework is designed to test out solutions for the MDRP. It also has a full unit test suite in place to provide support when shipping new code. The framework simulates the execution of the solution and outputs performance metrics, representing a real-life operation. Keeping this in

mind, potential users can focus on implementing models and algorithms (create policies), using the computational framework to compare results, without having to worry about the nuance of finding a way to apply their solution.

Figure 1 showcases how the computational framework is built. The framework is built on top of Python(Python Software Foundation, 2020). The *World* module advances the internal clock. For each time stamp generated, it queries a PostgreSQL(The PostgreSQL Global Development Group, 2020) data base (DDBB) that is mounted via a Docker(Docker Inc, 2020) container. The DDBB contains the data instances where it is known when a user or courier logs on to the system and their respective properties, including the order associated to the user. If there is new information to be retrieved, the query returns the users and/or couriers. This data is pushed to the *Simulator* module

The *Simulator* module is built on top of SimPy(Team SimPy, 2020). This simulator recreates the entire logic described in the **problem description**. After a simulation is completed, the results and output are written to the local *PostgreSQL* DDBB via *Docker*. The simulator uses policies to help actors execute actions or make decisions. These policies are modularized and conceived as building blocks: easily interchangeable and flexible for testing different models and algorithms in the framework. The policies are the units that make use of the services available inside the computational framework. Currently, the *Simulator* makes use of two services: the *Optimizer* and a *OSRM* (Open Source Routing Machine)(OSRM, 2020) container.

The *Optimizer* service provides an interface to solve mathematical optimization models with COIN-OR(COIN OR Foundation, 2020) or Gurobi(Gurobi Optimization, LLC, 2020). The *OSRM* service is mounted via *Docker* and receives requests. A request contains an origin, a destination and a vehicle. Each location is characterized by a (lat, lng) tuple. The service returns the city travel time it takes to go from that origin to that destination, using the vehicle provided in the request.

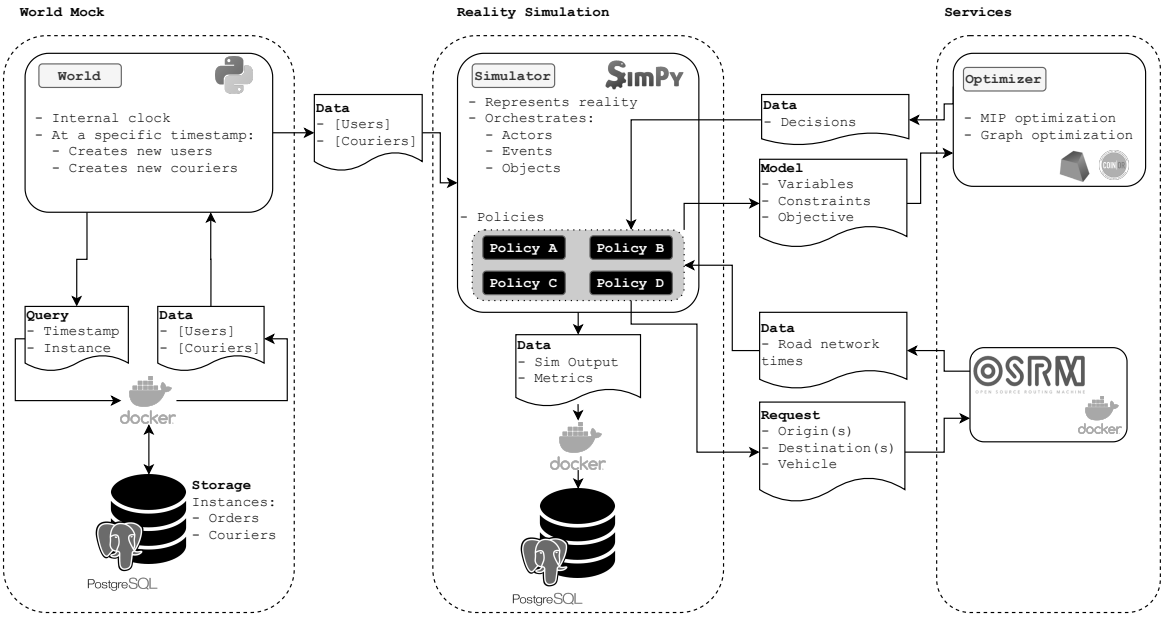


Figure 1: Computational framework built around three modules.

The code base for the computational framework can be found in the mdrp-sim Github repository(Quintero, 2020). The reader is encouraged to visit the repository and follow the README to execute a simulation. Having explained what the computational framework is, this chapter further explains the design of the simulator and policies used therein.

5.2 Discrete Events Simulator for the MDRP

The simulator accurately and robustly represents an operation governed by the **problem description**. These are the components of the simulator:

- *Event*. A sequence of items, facts, actions or changes triggered at a moment in time that follow a chronological order.
- *Actor*. Entity that makes decisions and executes actions (triggers events).
- *State*. Current condition of an actor. Can also be defined as a sequence of events during a specific time interval pertaining to an actor.
- *Policy*. Algorithms, models, logic and general conditions that structure how an actor makes a decision or executes an action.
- *Object*. Passive entity used to represent an abstract object, person or place. Does not make decisions or execute actions.

The simulator has a *start time* and *end time*. Although the available instances in this research are made up of a complete day (0 h - 23:59 h), a simulation can be performed over any partial time window. Moreover, the simulator has a *warm-up time* to achieve steady state and any events generated during the warm-up period are discarded. To provide flexibility for the simulation time window, there are constants that allow the handling of new entities entering the system: *create users from*, *create users until*, *create couriers from* and *create couriers until* times. These times can serve to provide a cool-down period to depart from the steady state.

The objects of the simulator are:

- *Order*. The main object of the simulation. It is a placeholder that registers events, as it is passed between the actors. It contains the attributes described in the **MDRP** and **novel characteristics** sections.
- *Location*. Physical place represented by a (lat, lng) tuple.
- *Notification*. Structure that relates a courier to an instruction that needs to be carried out.
- *Stop*. Object characterized by having a location and time that may contain orders to be picked up or dropped off.
- *Route*. An ordered set of stops that may contain orders to be delivered.
- *Vehicle*. Transportation mode of a courier.

There are three actors in the simulator:

1. *User*. A user $u \in U$ who orders meals.
2. *Courier*. A courier $c \in C$ who delivers meals.
3. *Dispatcher*. The system controller. It may also be referred to as "the company". This actor is responsible for the operation and for fulfilling orders. It orchestrates the communication between the user and the courier.

The restaurant is not modeled as an actor as no decisions are made from their part. It is assumed that all placed orders are cooked and the ready time is always fulfilled. However, the restaurant may be represented as an actor in future research where decisions, such as if an order is accepted, or the time it takes to prepare a meal, are taken.

In the following sections, the events, states and policies of each actor are described. Given that the policies contain the more complex logic and algorithms, all the available policies are separately described afterwards.

5.2.1 User

Figure 2 contains the state transition diagram for the user. Since a user is linked to a single order and viceversa, we may refer to the properties of order $o \in O$ as belonging to the user u_o indiscriminately. The user logs on (is sent from the *World*) to the system and enters the *idle* state at placement time a_o. It immediately triggers the *submit order* event and enters the *waiting* state. At time $a_o + t^u$, the *evaluate cancellation* event is triggered and *cancellation* policy $\mathbb{P}_{uc}$ is evaluated. If the user decides to cancel the order, the *cancel order* event is triggered, sent to the dispatcher, and the user logs off of the system. On the other hand, if the user decides to wait for the order, it returns back to the *waiting* state. When the order is delivered, at time $f_o + s^u$, the *order dropped off* event is triggered, and the user logs off of the system. If the order is canceled by the dispatcher, the *cancel order* event is triggered.

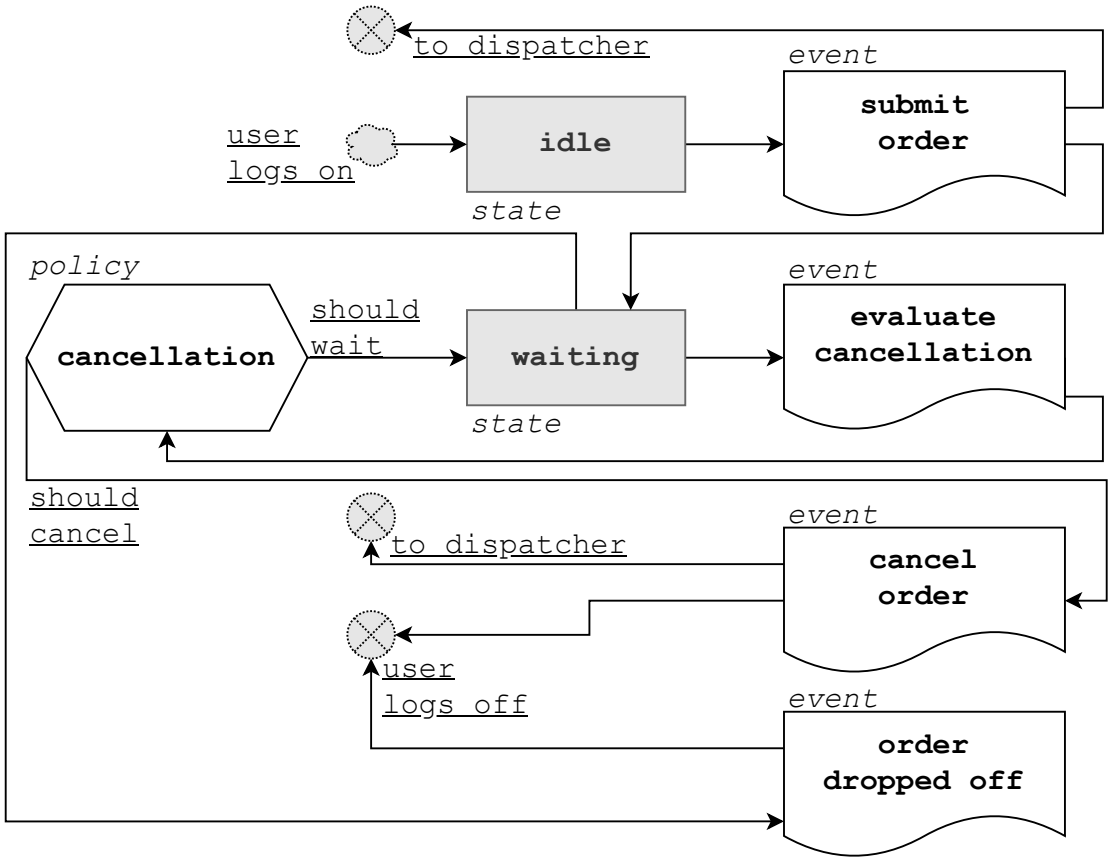


Figure 2: User state transition diagram.

Events

- *Submit order* event: details how the user submits a new order.
- *Evaluate cancellation* event: details how the user evaluates if it should cancel an order (incorporates *cancellation* policy).
- *Cancel order* event: details the actions taken by a user to cancel an order.
- *Order dropped off*: details the actions taken by a user when the order is dropped off.

States

- *Idle* state: passive actor state before submitting a new order.

- *Waiting* state: state in which the user is waiting for the order to be dropped off.

Policies

- *Cancellation* policy: $\mathbb{P}_{uc}$ establishes how a user decides to cancel an order.

5.2.2 Courier

Figure 3 contains the state transition diagram for the courier. The courier logs on from the world at time e_c and enters the *idle* state. At time l_c, the *log off* event is triggered and earnings are calculated. Apart from this event, there are two main event pipelines: the *evaluate movement* pipeline and the *notification* pipeline.

The *evaluate movement* pipeline starts when the *evaluate movement* event is triggered at every time fraction f_r. It leads to the *movement evaluation* policy $\mathbb{P}_{me}$. If the courier decides not to move, it goes back to the *idle* state. On the other hand, if the courier decides to move from $\ell_{c,t}$ to $\ell_{c,t+1}$, the *movement* event follows. It directs to the *moving* state. The movement mechanics are structured in the *movement* policy $\mathbb{P}_{mp}$. The pipeline ends with the courier going back to an *idle* state.

The *notification* pipeline starts with the *notification* event arriving from the *idle* state. Immediately, the courier decides to accept or reject the notification based on the *acceptance* policy $\mathbb{P}_{ap}$. Should the courier reject this notification, it goes back to being in an *idle* state. If the notification is accepted, the courier enters the aforementioned *moving* state. The courier moves to the pick-up location ℓ_r and enters the *picking up* state. After service time s^r, the courier enters the *moving* state. Alternatively, the courier may receive additional n_p notifications, in which case a partial *notification* pipeline is triggered where in case of acceptance or rejection, the courier is back at the *picking up* state. After arriving at the drop-off location ℓ_u, the *dropping off* state starts. When service time s^u ends, the courier moves to the other locations ℓ_u if $|s| > 0$, otherwise, the courier goes back to the *idle* state.

Events

- *Evaluate movement* event: details how the courier decides if it should move (incorporates the *movement evaluation* policy).
- *Move* event: details the actions taken by the courier to move somewhere (incorporates the *moving* state and *movement* policy).
- *Notification* event: details how a courier handles a new notification (incorporates the *acceptance* policy and subsequently the *picking up* and *dropping off* states).
- *Log off* event: details how the courier logs off of the system (includes earnings calculations).

States

- *Idle* state: stand-by state of the courier. After moving and executing notifications, the courier always comes back to this state.
- *Moving* state: establishes how the courier moves from an origin to a destination.
- *Picking up* state: details the actions taken by a courier for picking up orders.
- *Dropping off* state: details the actions taken by a courier for dropping off orders.

Policies

- *Movement evaluation* policy: $\mathbb{P}_{me}$ establishes how a courier decides whether and where it wants to relocate.
- *Movement* policy: $\mathbb{P}_{mp}$ establishes how the courier moves about the city and between route stops.
- *Acceptance* policy: $\mathbb{P}_{ap}$ establishes how a courier decides to accept or reject a notification.

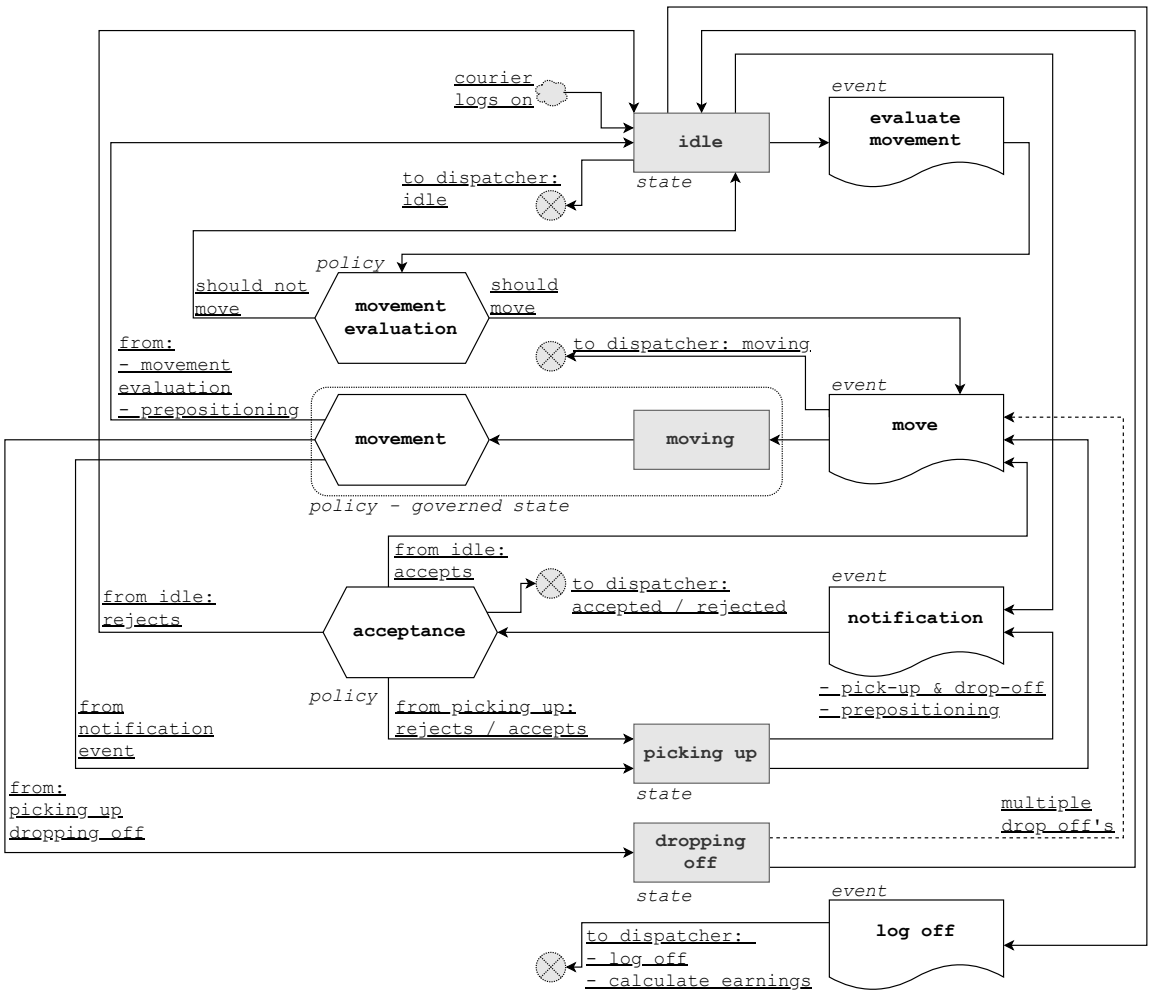


Figure 3: Courier state transition diagram.

5.2.3 Dispatcher

Figure 4 contains the state transition diagram for the dispatcher. The dispatcher is instantiated at the beginning of the *World* and enters a *listening* state. This is the only state, as the dispatcher's main task is orchestrating events. Some events that arrive are the *update / control* events, which are generalized events that trigger actions over an actor's state, but are self-contained and do not rely on policies or other events. For this reason, they are condensed into a single *update / control* event pipeline. The other three main pipelines of action for the dispatcher are: the *order submitted* pipeline, the *evaluate buffering* pipeline and the *evaluate prepositioning* pipeline.

The *order submitted* pipeline starts when a user submits a new order at placement time a_o and triggers the *order submitted* event. This event leads to the *buffer order* event, where, at preparation time d_o, the order is placed in the unassigned orders buffer. At time $d_o + t^u$ the *evaluate cancellation* event is triggered and dispatcher *cancellation* policy $\mathbb{P}_{dc}$ is evaluated. If the dispatcher decides to cancel the order, the *cancel order* event is triggered, along with the user's corresponding event; else, nothing further happens in this pipeline.

The *evaluate buffering* pipeline starts with the triggering of the *evaluate buffering* event at every time fraction f. *Buffering* policy $\mathbb{P}_b$ determines if a *dispatch* event should be triggered or the order should be buffered. When the *dispatch* event is triggered, the *matching* policy $\mathbb{P}_m$ is called upon, and consequently, several (or none) *notification* events (containing n_{pd} notifications) are triggered and sent to the courier.

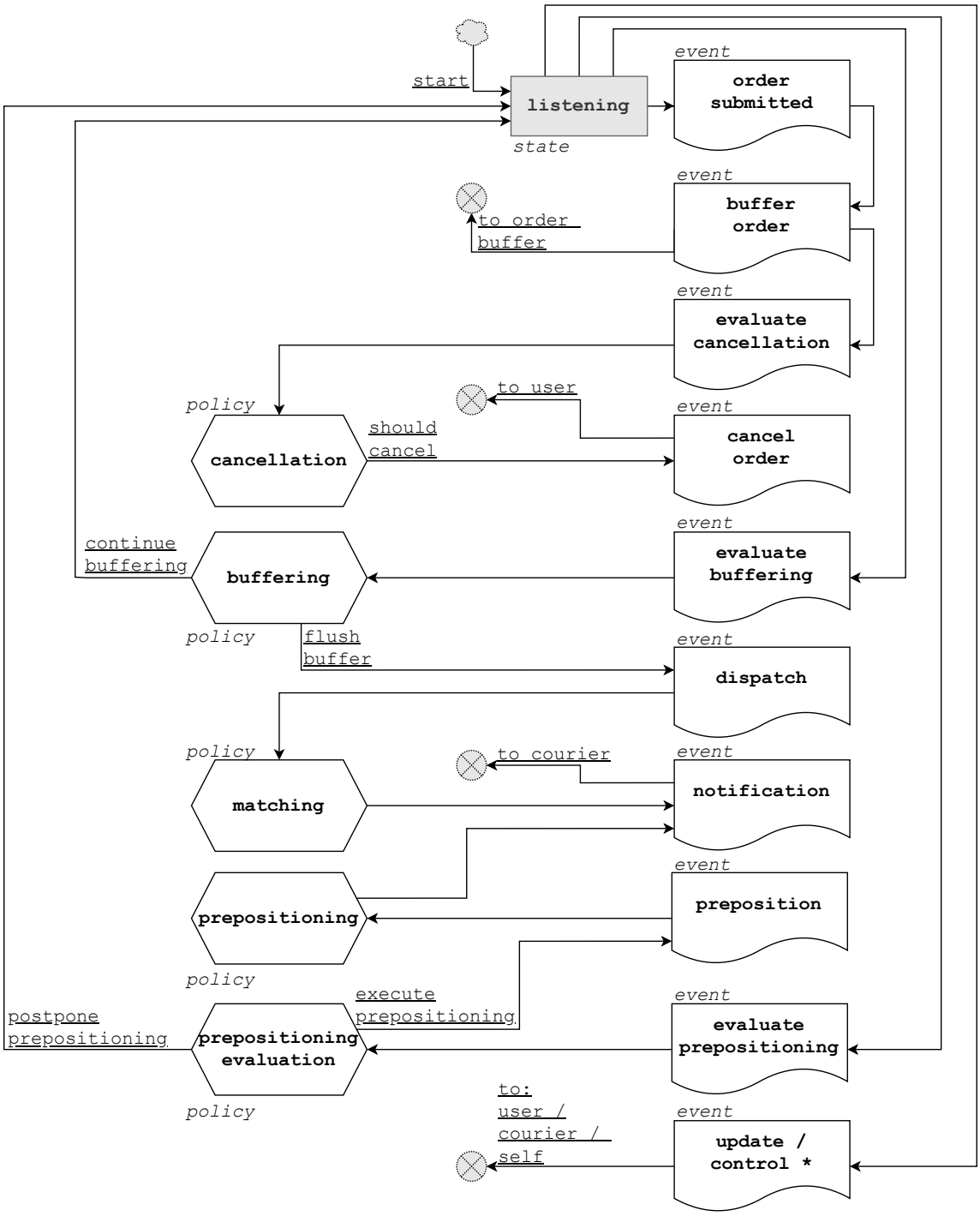
Lastly, the *evaluate prepositioning* pipeline starts when the *evaluate prepositioning* event is triggered at every time fraction f_p. The *prepositioning evaluation* policy $\mathbb{P}_{pe}$ determines if a *preposition* event is triggered. When this event is activated, the *prepositioning* policy $\mathbb{P}_p$ determines if *notification* events should be sent to the courier, carrying n_p notifications.

Events

- *Order submitted* event: details how the dispatcher handles the submission of a new order.
- *Buffer order* event: details how the dispatcher buffers a newly created order.
- *Evaluate cancellation* event: details the actions taken by the dispatcher to consider canceling an order (incorporates the *cancellation* policy).
- *Cancel order* event: steps that the dispatcher takes to cancel an order.
- *Evaluate buffering* event: establishes how the dispatcher evaluates if it should keep buffering orders or rather flush the buffer (incorporates the *buffering* policy and triggers the *dispatch* event).
- *Dispatch* event: actions taken by the dispatcher to route and match orders and couriers (incorporates the *matching* policy and triggers the *notification* event).
- *Notification* event: notify pick-up & drop-off notifications or prepositioning notifications to couriers.
- *Preposition* event: actions taken by the dispatcher to preposition couriers (incorporates the *prepositioning* policy and triggers the *notification* event).
- *Evaluate prepositioning* event: details the actions taken by the dispatcher to consider executing a prepositioning pipeline.
- *Update / Control* event: the dispatcher has the faculty of controlling an updating the system based on what is happening. As such, it is in charge of updating user, order and courier properties.

States

- *Listening* state: the sole state of the dispatcher, since the role of this actor is to react and coordinate system changes.



***update / control events:** notification accepted, notification rejected, orders in store, orders picked up, orders dropped off, courier idle, courier moving, courier picking up, courier dropping off, courier logs off*

Figure 4: Dispatcher state transition diagram.

Policies

- *Cancellation* policy: $\mathbb{P}_{dc}$ establishes how the dispatcher decides to cancel an order.
- *Buffering* policy: $\mathbb{P}_{b}$ establishes how the dispatcher buffers orders before dispatching them, this is, when the dispatcher flushes the unassigned buffer.
- *Matching* policy: $\mathbb{P}_{m}$ establishes how the dispatcher executes routing and matching between orders and couriers.
- *Prepositioning* policy: $\mathbb{P}_{dp}$ establishes how the dispatcher executes prepositioning of couriers.
- *Prepositioning evaluation* policy: $\mathbb{P}_{pe}$ establishes how the dispatcher decides if prepositioning should be done and how often.

5.3 Policies

The groundwork of the problem's mechanics is the main contribution of this research, as it was explained in the **computational framework**. Returning to the definition of the **simulator**, policies are the algorithms and models that govern how a certain process or decision is executed. Policies can be swapped around and different instances can be tested against different policies. For example, for the user *cancellation* policy $\mathbb{P}_{uc}$, option A can be tested. After a few runs, it can be changed in favor of option B.

The notation for a policy $\mathbb{P}$ is defined as:

$$\mathbb{P}_{policy}(I_{policy}) \rightarrow P'_{policy} \tag{1}$$

where *policy* is the name of the policy, I is the set of inputs $|I| \geq 1$ and P' is the output $|P'| = 1$. A policy $\mathbb{P}$ can contain any type of logic: from an absolute rule up to an optimization algorithm. Extending the definition of Equation (1), the policies for the **user**, **courier** and **dispatcher** are structured by definitions 5.1 through 5.9.

Definition 5.1 (User Cancellation Policy)**.** The user decides if they want to cancel the order. Inputs: courier assigned to the order. Output: binary decision.

$$\mathbb{P}_{uc}(c_o) \rightarrow \{0, 1\} \tag{2}$$

Definition 5.2 (Courier Movement Evaluation Policy)**.** The courier decides whether and where they want to move. Inputs: current location of the courier. Output: desired destination for the courier, if any.

$$\mathbb{P}_{me}(\ell_{c,t}) \rightarrow \ell_{c,t+1} \cup \emptyset \tag{3}$$

Definition 5.3 (Courier Movement Policy)**.** Mechanics for moving the courier in the city. Inputs: origin, destination and vehicle. Output: none.

$$\mathbb{P}_{mp}(\ell_1, \ell_2, v_c) \rightarrow \emptyset \tag{4}$$

Definition 5.4 (Courier Acceptance Policy)**.** The courier decides if they want to accept a notification. Inputs: acceptance rate, haversine distance to the notification. Output: binary decision.

$$\mathbb{P}_{ap}(p_c, h_{\ell_{c,t},\ell_n}) \rightarrow \{0, 1\} \tag{5}$$

Definition 5.5 (Dispatcher Cancellation Policy)**.** The dispatcher decides if they want to cancel the order. Inputs: courier assigned to the order. Output: binary decision.

$$\mathbb{P}_{dc}(c_o) \rightarrow \{0, 1\} \tag{6}$$

Definition 5.6 (Dispatcher Buffering Policy)**.** The dispatcher decides if the order should be buffered or a dispatch event be triggered otherwise. Inputs: present time. Output: binary decision.

$$\mathbb{P}_b(t) \rightarrow \{0, 1\} \tag{7}$$

Definition 5.7 (Dispatcher Matching Policy)**.** The dispatcher matches orders to couriers. Inputs: orders, couriers and the present time. Output: notifications, if any.

$$\mathbb{P}_m(O, C, t) \rightarrow \{n_{pd},\ \forall\ n_{pd} \in N\} \cup \emptyset \tag{8}$$

Definition 5.8 (Dispatcher Prepositioning Policy)**.** The dispatcher prepositions couriers. Inputs: orders, couriers and the present time. Output: prepositioning notifications, if any.

$$\mathbb{P}_{dp}(O, C, t) \rightarrow \{n_p,\ \forall\ n_p \in N\} \cup \emptyset \tag{9}$$

Definition 5.9 (Dispatcher Prepositioning Evaluation Policy)**.** The dispatcher decides if they want to trigger a prepositioning event. Inputs: present time. Output: binary decision.

$$\mathbb{P}_{pe}(t) \rightarrow \{0, 1\} \tag{10}$$

Having formalized the general type of policies available in the computational framework, the particular policies are introduced. A particular policy $\mathbb{P}'$[`particular`] implements the function or logic that allows for $I \rightarrow P'$. In other words, $\mathbb{P}'$ is the implementation of $\mathbb{P}$, such that:

$$\mathbb{P}'_{policy}[\text{particular}](I_{policy}) \rightarrow P'_{policy} = f(...) \tag{11}$$

where $f(...)$ is the actual implementation of the `particular` policy. Several particular policies are implemented in this research to showcase how they leverage on the computational framework to provide solutions around the MDRP.

5.3.1 User Policies

For the user *cancellation* policy, a `random` policy is defined by Equation (12):

$$\mathbb{P}'_{uc}[\text{random}](c_o) \rightarrow \{0, 1\} = \begin{cases} X \sim U(0,1) \geq x_u & c_o = \emptyset \\ 0 & c_o \neq \emptyset \end{cases} \tag{12}$$

where a random variable X is compared against the user cancellation probability x_u. The user randomly decides if they want to cancel the order as long as no courier is assigned.

5.3.2 Courier Policies

The courier's *acceptance* policy is proposed as an `absolute` policy (where all notifications are accepted) in Equation (13) or a `uniform` policy in Equation (14), where the uniform distribution is evaluated to determine is a courier accepts a notification.

$$\mathbb{P}'_{ap}[\text{absolute}](c_o) \rightarrow \{0, 1\} = 1 \tag{13}$$

$$\mathbb{P}'_{ap}[\text{uniform}](c_o) \rightarrow \{0, 1\} = \begin{cases} 1 & X \sim U(0,1) \leq p_c \\ 0 & \text{otherwise} \end{cases} \tag{14}$$

where a random variable X that is distributed uniformly is compared against the courier's acceptance rate p_c. The courier randomly decides if they want to accept the notification, otherwise, it is rejected.

To realistically move about the city, a `osrm` *movement* policy $\mathbb{P}'_{mp}[\text{osrm}]$ is proposed. The *OSRM* (Open Source Routing Machine)(OSRM, 2020) service is used, as showcased in Figure 1. The service receives a request containing an origin ℓ_1, a destination ℓ_2 and a vehicle $v \in V$. The origin and destination are (lat, lng) tuples. The docker-mounted server returns a series of waypoints, corresponding to the intersections in the city that make up the route. For each leg of the route (pair of intersections) ℓ'_1 to ℓ'_2, a haversine distance is calculated $h_{\ell'_1,\ell'_2}$. This is shown in Figure 5. This distance is then divided by the default velocity of a vehicle $v \in V$, to obtain the time it would take to complete that leg of the journey. If all the times are added together, it results in an approximation to how long it would take to realistically move from an origin to a destination. This time is used to move couriers while time in the simulator advances. In Figure 5, this means that $b_{A,F} = \frac{h_{A,B}+h_{B,C}+h_{C,D}+h_{D,E}+h_{E,F}}{v_{velocity}}$.

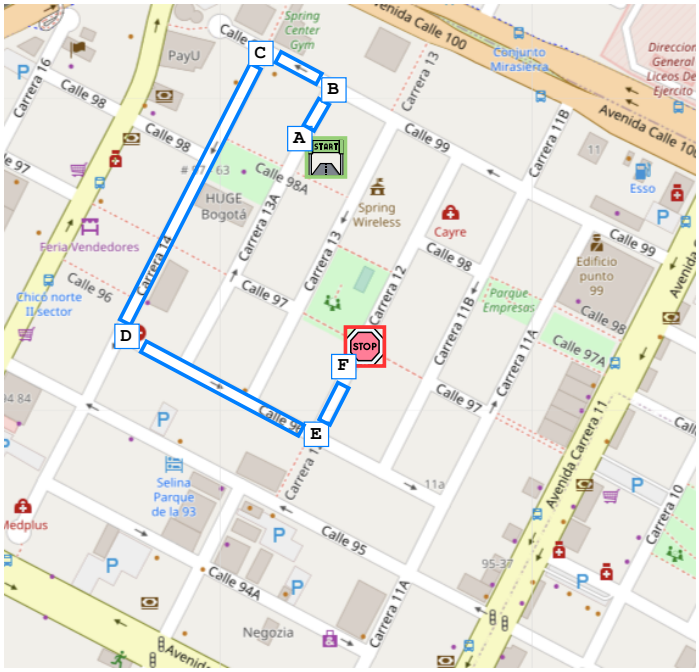


Figure 5: Example of how the OSRM service calculates movement times.

The last policies defined for the courier are the *movement evaluation* policies. The `still` and `neighbors` policies are proposed. The `still` *movement evaluation* policy is shown in Equation (15) and all it says is the courier always decides to stay put.

$$\mathbb{P}'_{me}[\text{still}](\ell_{c,t}) \rightarrow \ell_{c,t+1} \cup \emptyset = \emptyset \tag{15}$$

On the other hand, the `neighbors` *movement evaluation* policy is explained in Algorithm 1 and is

based around the concept of geohash and geohash neighborhoods.

Algorithm 1: Geohash neighbors heuristic

Policy: $\mathbb{P}'_{me}$[neighbors]
Input : $(\ell_{c,t})$
Output: $\ell_{c,t+1} \cup \emptyset$

if *random* $\leq$ *movement probability* **then**
  geohash $\leftarrow$ geoEncode($\ell_{c,t}$, precision $= 6$)
    ... Obtain a geohash from the courier location with a level 6 precision
  geohashNeighbors $\leftarrow$ geoNeighbors(geohash) ... Obtain a neighborhood of 8 geohashes
  chosenGeohash $\leftarrow$ choose{geohashNeighbors} ... Randomly select the destination geohash
  $\ell_{c,t+1} \leftarrow$ geoDecode{chosenGeohash}
    ... Obtain a location inside the chosen geohash like its center
else
  $\ell_{c,t+1} \leftarrow \emptyset$ ... If the courier decides not to move, then his destination is null
end
return $\ell_{c,t+1}$

Based on a certain movement probability for the courier, this is, if the courier decides to move, a neighborhood of geohashes is generated. From that neighborhood, one of the geohashes is randomly chosen. A location ℓ inside that geohash is obtained and selected as the destination for the courier. If the courier decides not to move, Algorithm 1 returns an empty destination. Recall that a courier decides to move at every f_r time fraction.

5.3.3 Dispatcher Policies

For the dispatcher *cancellation* policy, a `static` policy is defined by Equation (16). If no courier is assigned to the order, the order is immediately canceled.

$$\mathbb{P}'_{dc}[\text{static}](c_o) \rightarrow \{0,1\} = \begin{cases} 1 & c_o = \emptyset \\ 0 & c_o \neq \emptyset \end{cases} \tag{16}$$

The dispatcher *buffering* policy has a `rolling-horizon` implementation defined by Equation (17). The dispatcher buffers orders until f and consequently flushes the unassigned buffer.

$$\mathbb{P}'_{b}[\text{rolling-horizon}](t) \rightarrow \{0,1\} = \begin{cases} 1 & t \mod f = 0 \\ 0 & t \mod f \neq 0 \end{cases} \tag{17}$$

Following the same approach as Equation (17), there is a `fixed` dispatcher *prepositioning evaluation* policy defined in Equation (18). The dispatcher waits until f_p to apply the *prepositioning* policy $\mathbb{P}_{dp}$.

$$\mathbb{P}'_{pe}[\text{fixed}](t) \rightarrow \{0,1\} = \begin{cases} 1 & t \mod f_p = 0 \\ 0 & t \mod f_p \neq 0 \end{cases} \tag{18}$$

A straight forward *prepositioning* policy $\mathbb{P}_{dp}$ is a `naive` approach, where simply no prepositioning notifications are sent, as depicted in Equation (19).

$$\mathbb{P}'_{dp}[\text{naive}](t) \rightarrow \{n_p,\ \forall\ n_p \in N\} \cup \emptyset = \emptyset \tag{19}$$

The *matching* policies proposed are more complex and thus are organized in a separate section.

5.3.3.1 Dispatcher Matching Policies

To have a baseline to which other *matching* policies can be compared against, a simple `greedy` *matching* policy $\mathbb{P}'_m$[greedy] is implemented in Algorithm 2.

Algorithm 2: Greedy matching algorithm

Policy: $\mathbb{P}'_m$[greedy]
Input : (O, C, t)
Output: $\{n_{pd},\ \forall\ n_{pd} \in N\} \cup \emptyset$

NotifiedCouriers $\leftarrow \{\}$ ... Initialize empty notified couriers
$N_t \leftarrow \{\}$ ... Initialize empty notifications
for $o \in O$ **do**
  SelectedCourier $\leftarrow \arg \min_{c \in C : c \notin \text{NotifiedCouriers}} \{b_{\ell_{c,t}, \ell_{r_o}}\}$ ... Select the minimum cost courier
  if *SelectedCourier* $\notin N_t \cup$ *NotifiedCouriers* **then**
    NotifiedCouriers $\leftarrow$ NotifiedCouriers $\cup$ {SelectedCourier} ... Cross out notified courier
    create n_{pd} ... Create a new notification
    $c_{n_{pd}} \leftarrow$ SelectedCourier ... Set selected courier to new notification
    $s_{n_{pd}} \leftarrow \{o\}$ ... Set order as a single route to new notification
    $N_t \leftarrow N_t \cup \{n_{pd}\}$ ... Append new notification
  end
end
return N_t

For each order, the minimum-cost courier that can be matched is selected and excluded from the available couriers. The notification is saved and the next order is processed, until all orders are matched or no couriers remain available. $\mathbb{P}'_m$[greedy] should be executed as a rolling horizon strategy, with $f > 0$.

For more elaborate *matching* policies, orders are first bundled into routes and these routes are matched to couriers. To calculate routes, a target route size Z_t is defined by Equation (20), adapted from Reyes et al. (2018).

$$Z_t = \max \left[\left\lceil \frac{|\{o \in O_t : e_o \leq t + \Delta_1\}|}{|\{c \in C_t : s_c = \emptyset\}|} \right\rceil, S_{max} \right], \Delta_1 > 0 \tag{20}$$

where O_t are the unassigned orders at time $t \in T$, C_t are the idle and picking-up couriers at time $t \in T$, and Δ_1 is set through any tuning procedure. The target route size Z_t defined by Equation (20)

is used in Algorithm 3 to calculate the set of routes S_t at time $t \in T$.

Algorithm 3: Routes generation using parallel-insertion

Input : O_t, C_t, Z_t
Output: S_t

for $r \in R_t$ ... *For each restaurant* **do**
  $O_r \leftarrow \{o, \forall o \in O_t : r_o = r\}$ ... Obtain orders for the restaurant
  $O_r^* \leftarrow \{O_r^i, \forall i \in |O_r| - 1 : e_{O_r^i} \leq e_{O_r^{i+1}}\}$ ... Sort orders by increasing ready time
  $C_r \leftarrow \{c \in C_t : h_{\ell_c, \ell_r} \leq d\}$
    ... Obtain couriers linked to the restaurant, based on a linear distance
  $m_r \leftarrow \max\left(|C_r|, \left\lceil \frac{|O_r^*|}{Z_t} \right\rceil\right)$ ... Obtain number of routes to create
  $S_r \leftarrow \{\emptyset, \forall i = 1 \dots m_r\}$ ... Initialize empty routes for the restaurant
  if *assignment updates allowed* **then**
    $C_r^* \leftarrow \{c \in C_r : s_c \neq \emptyset \ \&\ r_o = r, \forall o \in s_c\}$
      ... Obtain couriers with non-empty routes whose pick-up location is the restaurant
    $S_r \leftarrow S_r \cup \{s_c, \forall c \in C_r\}$
      ... Append courier routes to the empty routes to be able to update the courier's assignments
  end
  for $o \in O_r^*$ **do**
    Find the route $s \in$
      S_r and the insertion position i_s for the order that minimizes the increase in route cost:
      $\sum_{(\ell_1, \ell_2) \in s} b_{\ell_1, \ell_2} + \sum_{\ell \in s} s^\ell$
    if *Insertion decreases route efficiency* **then**
      Disregard s for order o and find the next best route and insertion position
    end
    $s^{i_s} \leftarrow o$ ... insert order o in route s at position i_s
  end
end
return S_t

where d is the maximum linear distance a courier can be away from a restaurant, m_r is the number of routes to create at restaurant $r \in R$, and s^ℓ is the service time at a location ℓ (s^r or s^u). Algorithm 3 is based on Procedure 1 of Reyes et al. (2018) but modified to include assignment updates if desired, although the original heuristic can be executed. With the routes S_t obtained in Algorithm 3, two linear assignment models are defined: an integer program in Definition 5.10 and a network flow formulation in Definition 5.11. The cost function for the assignment models, this is, $g_{c,s}$: if route s where to be assigned to courier c, is shown in Equation (21) and uses the following notation (Reyes et al., 2018):

- θ: penalty for a delayed drop-off.
- $\pi_{s,c}$: pick-up time of route s if assigned to courier c. $\pi_{s,c} \geq \max_{o \in s}\{e_o\}$.
- $\delta^s_{o,c}$: drop-off time of order o in route s if assigned to courier c.

$$g_{c,s} = \frac{|s|}{\max_{o \in s}\{\delta^s_{o,c}\}} - \theta\left(\pi_{s,c} - \max_{o \in s}\{e_o\}\right) \tag{21}$$

Definition 5.10 (Integer assignment model)**.**

$$\max \sum_{c \in C_t} \sum_{s \in S_t} g_{c,s} x_{c,s} \tag{22}$$

$$\text{s.t.} \sum_{s \in S_t} x_{c,s} \leq 1, \ \forall c \in C_t \tag{23}$$

$$\sum_{c \in C \cup \{0\}} x_{c,s} = 1, \ \forall s \in S_t \tag{24}$$

$$x_{c,s} \in \{0,1\}, \ \forall s \in S_t, c \in C_t \cup \{0\} \tag{25}$$

where $x_{c,s}$ is a binary decision variable that establishes if courier $c \in C_t$ is assigned to route $s \in S_t$. There is an artificial courier 0 that collects excess routes. The value of such assignments is 0.

Definition 5.11 (Network flow assignment model)**.**

$$\max \sum_{(i,j) \in A^*} c^*_{i,j} x_{i,j} \tag{26}$$

$$\text{s.t.} \sum_{j \in N^*} x_{i,j} - \sum_{j \in N^*} x_{j,i} = b^*_i, \ \forall i \in N^* \tag{27}$$

$$x_{i,j} \geq 0, \ \forall (i,j) \in A^* \tag{28}$$

where A^* is the set of arcs, N^* is the set of nodes, $c^*_{i,j}$ is the matching cost ($g_{c,s}$ for assignment arcs), b^*_i is the demand of node $i \in N^*$, and $x_{i,j}$ is the variable that captures the flow through the arcs in the network. There is a supply node whose demand is $|S_t|$. There is a node for each courier $c \in C_t$ with a demand of 0. There is a node for each route $s \in S_t$ with a demand of -1. Arcs depart from the supply node to the courier nodes with a cost of 0. In addition, there are arcs that depart from the supply and go toward the route nodes. These arcs activate when a route could not be matched to a courier and thus have a cost of zero. Arcs that depart from the courier nodes and arrive at the route nodes have the cost structure given in Equation (21).

Calculating every possible combination when matching is disadvantageous, given that many prospective matches can de discarded beforehand. For example, a courier who is on the other side of the city with respect to an order can be discarded, as this match will never get chosen (from the service side this will result in a terrible customer experience) and adds unnecessary complexity to Definitions 5.10 and 5.11. If prospects are incorporated in the matching policy, then Definition 5.12 should be coupled.

Definition 5.12 (Matching prospects)**.** Only create a matching variable $x_{c,s}$ for courier $c \in C$ and route $s \in S$, if all the following conditions are met:

$$h_{\ell_c, \ell_r} \leq d \tag{29}$$

$$|\pi_{s,c} - \max_{o \in s}\{e_o\}| + \sum_{o \in s} |f_o{}' - \delta^s_{o,c}| \leq \beta \tag{30}$$

$$s_c \subset s : s_c \neq \emptyset \tag{31}$$

Equation (29) establishes that a courier can not be very far away from the restaurant. Equation (30) uses β as a maximum allowed total off-set from the stops' expected time. For pick-up stops, the expected time is the ready time e_o. For drop-off stops, the expected time is the user's expected drop-off time f'_o. Equation (31) establishes that if an order was inserted into a courier's route (an assignment update) then the newly formed route must correspond to the courier. If these conditions are met, then the matching variable can be created.

The last aspect of creating new *matching* policies departing from the routes, matching prospects and exact matching is processing each tentative assignment $n_{pd} \in N_t$ with a commitment strategy that dictates (Reyes et al., 2018):

1. If c_n can arrive at ℓ_n before $t+f$ and all orders in s_n are ready by $t+f$ then keep n as a pick-up & drop-off notification (n_{pd}).
2. If c_n cannot reach ℓ_n by $t+f$ and is idle then transform n to a prepositioning notification n_p. This is done to have a backup in case no other best courier is found.
3. If any order in s_n has been ready for a long time (loosely defined), keep the notification and execute it immediately.

With this commitment strategy, the system opts to keep service quality as high as possible having couriers arriving just in time at the pick-up locations.

Combining the calculation of routes from Algorithm 3, the matches obtained from model 5.10 or 5.11, the definition of prospects from Definition 5.12, and the commitment strategy, a myopic *matching* policy is defined in Algorithm 4. This myopic *matching* policy should be executed as a rolling horizon approach, being processed every f fraction of time, $f > 0$.

Algorithm 4: Myopic matching policy

Policy: $\mathbb{P}'_m[...]$
Input : (O, C, t)
Output: $\{n_{pd},\ \forall\ n_{pd} \in N\} \cup \emptyset$

$S_t \leftarrow$ Generate routes with or without assignment updates
$\{x_{c,s},\ \forall s \in S_t, c \in C_t\} \leftarrow$ Generate matching prospects based on conditions
$N_t \leftarrow$ Generate matches (notifications) using an assignment model
for $n \in N_t$ **do**
  Execute commitment strategy
end
return N_t

Considering Algorithm 4, several *matching* policies are defined in Table 1.

Table 1: Definition of myopic matching policies

Policy	Assignment Updates	Prospects	Matcher	Commitment Strategy
$\mathbb{P}'_m$[mdrp]			Integer	✓
$\mathbb{P}'_m$[mdrp-graph]			Network Flow	✓
$\mathbb{P}'_m$[mdrp-graph-prospects]		✓	Network Flow	✓
$\mathbb{P}'_m$[modified-mdrp]	✓	✓	Network Flow	✓

Note that $\mathbb{P}'_m$[mdrp] is an implementation of Reyes et al. (2018).

6 Results and Discussion

6.1 Instances

24 real-life instances are provided by the company Rappi. Each instance is a complete day of operation (12 a.m. to 11:59 p.m.) for a specific city. The instances span six cities, thus providing four different scenarios for each city considered. Figure 6 shows the sizes of each instance and their classification as a small (S), medium (M) or large (L) instance. In Table 2, the different instances are classified by size. These real and large instances provide a clear opportunity for academic research, due to the size and quality of data.

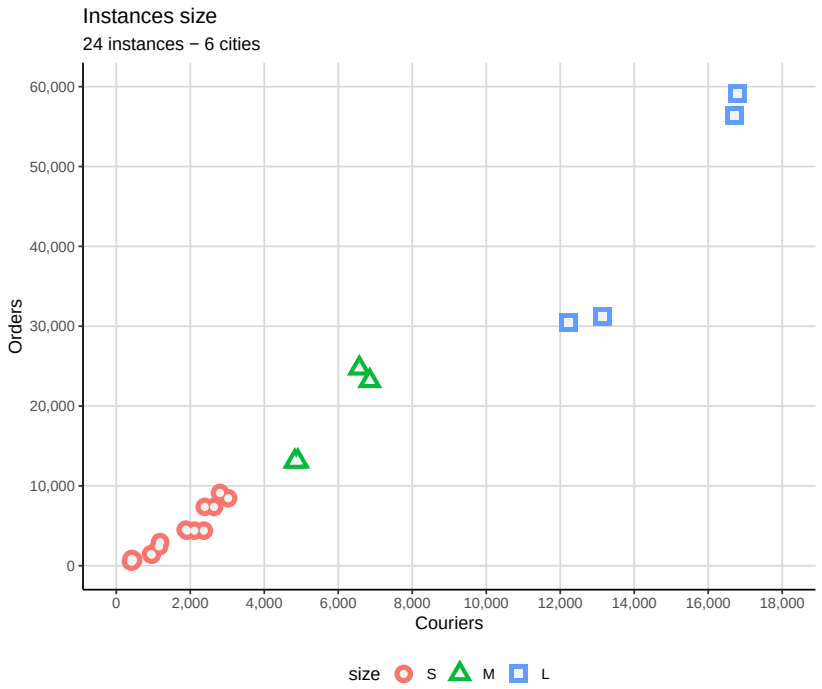


Figure 6: Number of orders and couriers provided in the different instances.

Table 2: Classification of instances

Size	Instances
S	1, 3, 4, 5, 7, 9, 10, 11, 13, 15, 16, 17, 19, 21, 22, 23
M	2, 8, 14, 20
L	0, 6, 12, 18

In the DDBB, the tables *couriers_instance_data* and *orders_instance_data* contain the information for the different instances. The *couriers_instance_data* table has information about all the couriers that came online for that day and is composed of the following columns:

- *courier_id*: an id to identify a courier.
- *vehicle* (v_c): mode of transportation. Can be: walking, bicycle, motorcycle or car.
- *on_lat* (part of ℓ_c): latitude of the courier's position at the start of the shift.
- *on_lng* (part of ℓ_c): longitude of the courier's position at the start of the shift.
- *on_time* (e_c): time stamp detailing when the courier started the shift.
- *off_time* (l_c): time stamp detailing when the courier ended the shift.

The *orders_instance_data* table contains information about all the orders processed during the day and is composed of the following columns:

- *order_id*: and id to identify an order.
- *pick_up_lat* (part of ℓ_r): latitude of the order's pick-up location.
- *pick_up_lng* (part of ℓ_r): longitude of the order's pick-up location.
- *drop_off_lat* (part of ℓ_u): latitude of the order's drop-off location.
- *drop_off_lng* (part of ℓ_u): longitude of the order's drop-off location.
- *placement_time* (a_o): time stamp detailing the order's creation time.

- *preparation_time* (d_o): time stamp detailing when the order starts being prepared.
- *ready_time* (e_o): time stamp detailing when the order is ready to be picked up.
- *expected_drop_off_time* (f'_o): time stamp detailing when the order should be dropped off.

Be aware that given the **structural assumptions**, a user can only place a single order per day (instance). Considering this, data pertaining orders can be referred to as belonging to users indiscriminately. In consequence, the arrival rates for users are the same as orders.

To understand how for different instance sizes the arrival rate of entities varies throughout the day, Figure 7 shows an aggregated count of the actors logging into the system per hour of the day. These figures show there is a significant difference between instance sizes, where the 16 small instances are as big as the four medium instances and these 20 instances are only about half as big as the four large instances. The behavior of the real-life arrivals into the system showcases there are two peaks for users ordering meals: lunch and dinner. Couriers understand these peaks, thus the majority start logging in some hours before the demand spikes.

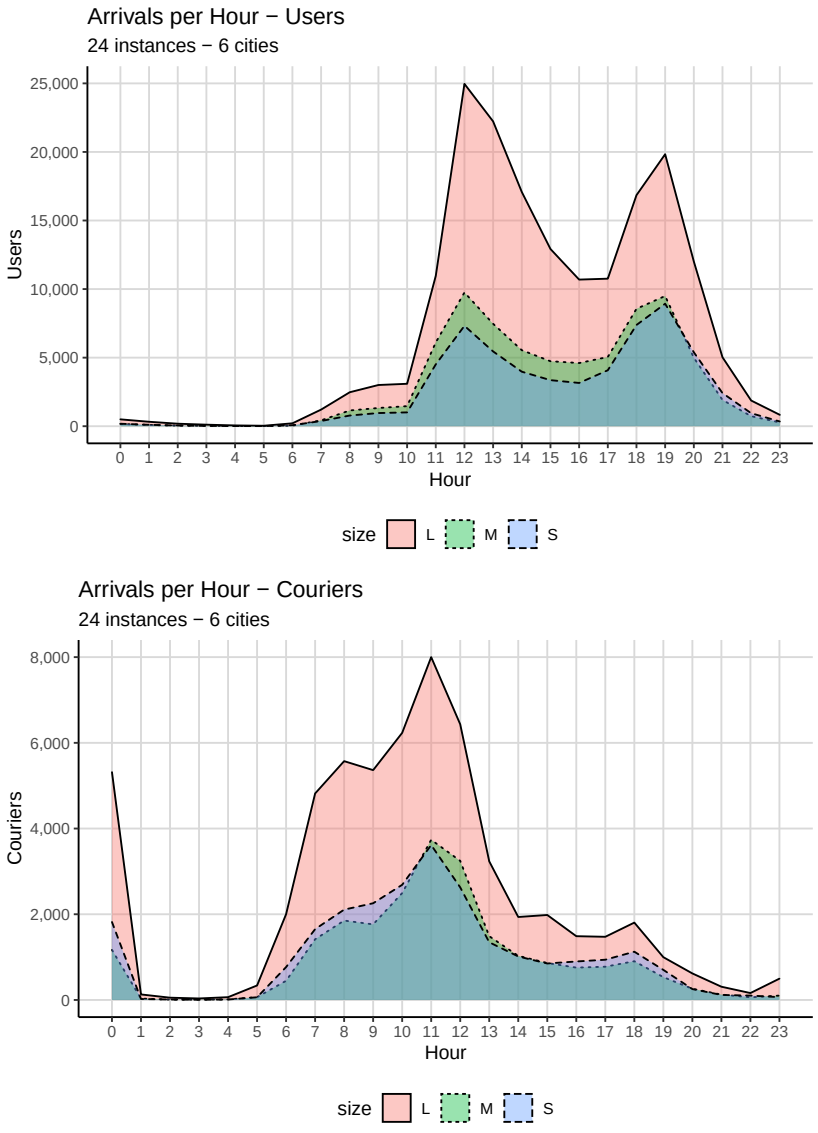


Figure 7: Aggregated entity arrivals by instance size.

6.2 Scenarios, Performance Metrics and Test Configurations

36 simulation scenarios are designed to test the computational framework. The complete list and description of scenarios can be found in Appendix B, along with the numerical values for all the variables used during the testing phase. Two experiments (operational configurations) are established, based on the courier's *acceptance* and *movement evaluation* policies.

- Experiment A (`absolute-still`): uses the `absolute` *acceptance* policy $\mathbb{P}'_{ap}$[absolute] and the `still` *movement evaluation* policy $\mathbb{P}'_{me}$[still]. These conditions are tested in Reyes et al. (2018) and Yildiz and Savelsbergh (2019).
- Experiment B (`uniform-neighbors`): uses the `uniform` *acceptance* policy $\mathbb{P}'_{ap}$[uniform] and the `neighbors` *movement evaluation* policy $\mathbb{P}'_{me}$[neighbors]. These conditions resemble a more life-like operation.

For each experiment, over the set of scenarios, the five proposed *matching* policies in Table 1 are run and the framework is used to compare which policy performs the best. The following performance metrics are used to compare the outcomes of applying different policies in the operation:

- *In store to pick-up time* (`store_pick`). Captures quality of assignments for the courier and system precision (just-in-time arrivals).
- *Ready to pick-up time* (`ready_pick`). Captures quality of service for both the user and courier by measuring the meals' loss of freshness.
- *Drop-off lateness time* (`lateness`). Captures quality of service for the user and quality of the solutions.
- *Click to taken time* (`click_taken`). Captures the effects of the prepositioning policy and the acceptance in the system.
- *Click to door time* (`click_door`). Captures quality of service for the user and system agility.
- *Fulfillment*. Captures quality of the solutions and service for all actors involved.

The results of the different simulations are saved in the DDBB in the tables *order_metrics*, *courier_metrics* and *matching_optimization_metrics*.

Three computers are used to run the simulations:

- MP: MacBook Pro, 2.3 GHz Quad-Core Intel Core i7, 16 GB 3733 MHz LPDDR4X Memory.
- MA: Macbook Air, 1.8 GHz Dual-Core Intel Core i5, 8 GB 1600 MHz DDR3 Memory.
- IM: iMac, 2.7 GHz Quad-Core Intel Core i5, 8 GB 1600 MHz DDR3 Memory.

The LINPACK benchmark (Dongarra et al., 2003) is used to compare the performance of the computers and calculated based on Appendix C.

- $MP = 116.8682$ GFlops s^{-1}
- $MA = 33.9102$ GFlops s^{-1}
- $IM = 47.6564$ GFlops s^{-1}

MP is the fastest computer, thus a factor is calculated using MP as reference to convert computational time across tests.

- $MA_F = 33.9102/116.8682 = 0.2902$
- $IM_F = 47.6564/116.8682 = 0.4078$

The `Gurobi` solver is used for testing (`COIN-OR` implementation with `pulp` is also available).

6.3 Results

6.3.1 An Ideal Operation

Figure 8 shows the result of experiment A (`absolute-still`) across all scenarios. Box plots describe the performance of the different *matching* policies, grouped by metric. A vertical line crosses zero for reference. The fulfillment rate is shown for each *matching* policy.

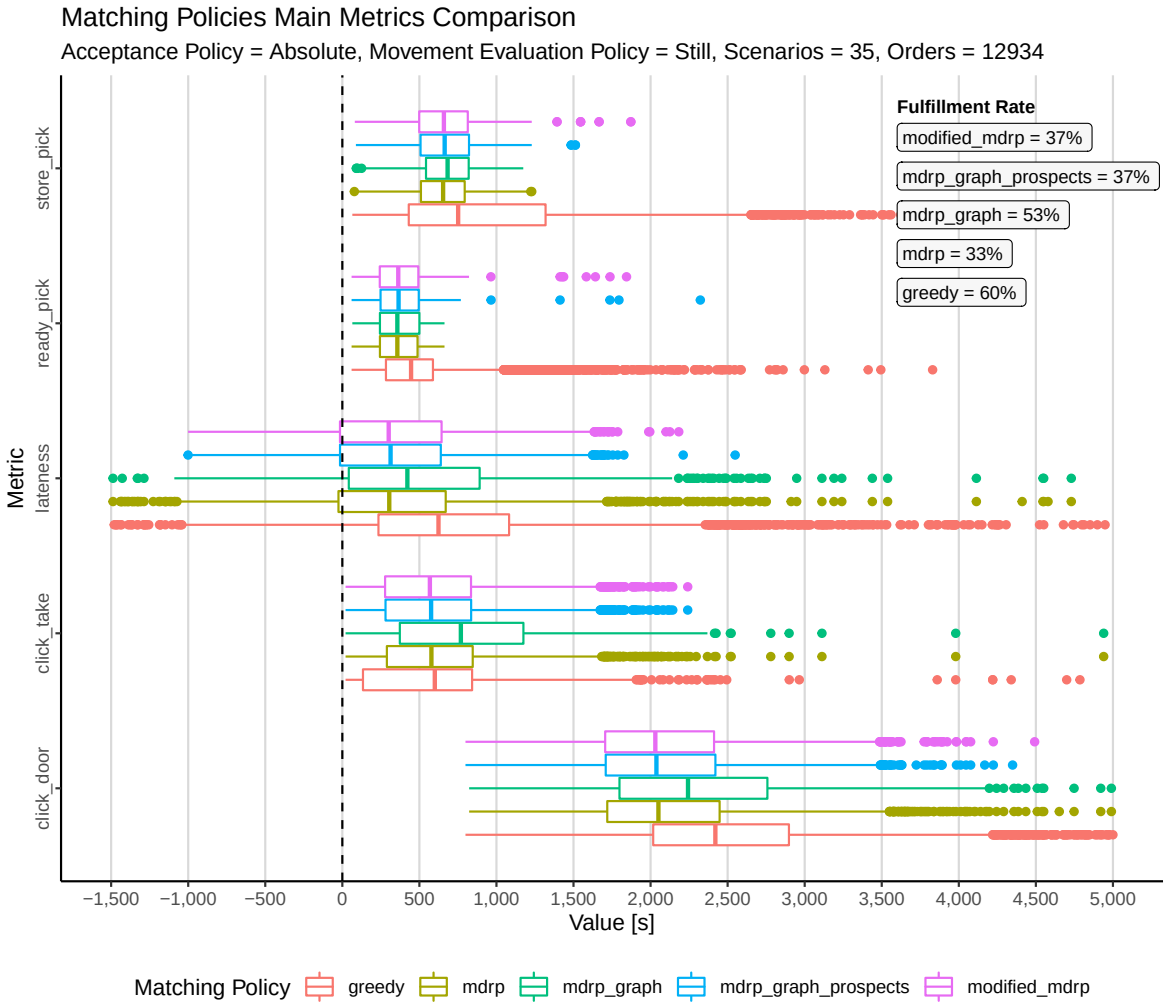


Figure 8: Performance metrics box plots comparison between matching policies for experiment A (absolute-still) across scenarios.

Regarding the five time-related metrics, the optimal behavior is to stand on the reference line. For fulfillment, a higher percentage is desired. The `greedy` policy shows the worse overall time-related performance: metrics are spread and median values are higher compared to the other policies. This behavior is expected due to the simplistic nature of the algorithm. Although the `greedy` policy shows the highest fulfillment (as orders are practically guaranteed to find a match disregarding the quality), service for these orders is not optimal. In real-life operations, many times it is best to prematurely cancel an order than to deliver it in poor conditions.

The four optimization *matching* policies show a very similar performance across time metrics. The `mdrp-graph` policy has a higher median for the drop-off lateness time, click to taken time and click

to door time. It can be seen that using the prospects strategy from Definition 5.12 reduces outliers, as the time distribution for the drop-off lateness time, click to taken time and click to door time for policies `mdrp-graph-prospects` and `modified-mdrp` are more adjusted when compared to the `mdrp` and `mdrp-graph` policies. There are few outliers for the `mdrp-graph-prospects` and `modified-mdrp` policies shown in the ready to pick-up time and in-store to pick-up time, which can be discarded, thus confirming that using the prospects strategy reduces unexpected edge cases.

With respect to the solver, using the network flow model from Definition 5.11 does not provide an edge over the MIP model from Definition 5.10 regarding the time metrics. In this operational configuration, the assignment updates strategy during the routing phase described in Algorithm 3 used by the `modified-mdrp` *matching* policy does not result in any observable advantage over the described metrics.

The fulfillment rate is greatly affected by the use of prospects, as the `mdrp-graph-prospects` and `modified-mdrp` policies have lower fulfillment rates compared to the `mdrp-graph` policy. In addition, the `mdrp` policy has the lowest acceptance rate, meaning in this ideal environment, using a network flow solver may provide an advantage for the fulfillment of orders.

Aggregating the performance of the different metrics, it is shown that introducing prospects in this "ideal" operation (couriers who accept all notifications and do not move about freely) improves the distribution of time-related metrics. The presence of assignment updates does not seem to influence the performance of the *matching* policy, given that the `mdrp-graph-prospects` and `modified-mdrp` policies have close to identical results.

6.3.2 A More Realistic Operation

Figure 9 shows the result of experiment B (`uniform-neighbors`) across all scenarios. Box plots describe the performance of the different *matching* policies, grouped by metric. A vertical line crosses zero for reference. The fulfillment rate is shown for each *matching* policy. This operational configuration resembles a more life-like operation, where couriers are able to move freely about the city and decide if they want to accept or reject notifications.

In a realistic setting, the `greedy` policy still achieves the best fulfillment. Moreover, it still performs the worst regarding all time metrics. At a glance, the other *matching* policies have similar medians, minima, maxima and inter-quartile ranges. A clear difference regarding Figure 8 is that in this operational condition, the `mdrp-graph` *matching* policy no longer under-performs compared to the other optimization policies. Once again, embedding the prospects strategy provides more predictable distributions with fewer outliers across metrics. The prospects strategy provides a better fulfillment rate, as it is evidenced by the 32% and 33% fulfillment of the `modified-mdrp` and `mdrp-graph-prospects` policies respectively. The `mdrp` policy is around 10 points worst, meaning the use of an MIP model hinders the solution space search.

Using a more intelligent model to execute matching and routing can improve delivery times by 7 minutes overall, as seen by the median difference in the click to door time between the optimization policies and the `greedy` *matching* policy. Although it falls short for the time metrics, this situation resembles how a company must make the decision of which model to use, given they optimize different metrics, leading to an important business question: what is more important? Delivering *more* orders or delivering *better* orders? Companies reluctant to invest in research should always keep in mind that better models can provide an edge on the competition. These results evidence that using the proposed framework can provide useful insights and support the decision process of the operations research involved in meal delivery.

The fulfillment rates shown in both experiments are very low given the definition of the `random` user and `fixed` dispatcher *cancellation* policies from Definitions 5.1 and 5.5 respectively. In these definitions, after a fixed time has passed from the preparation time, the decision is made to cancel an order. Orders with long preparation times are affected since an order can be in the preparation process and be canceled. An improvement to these definitions may be to set the time at some moment *before* the

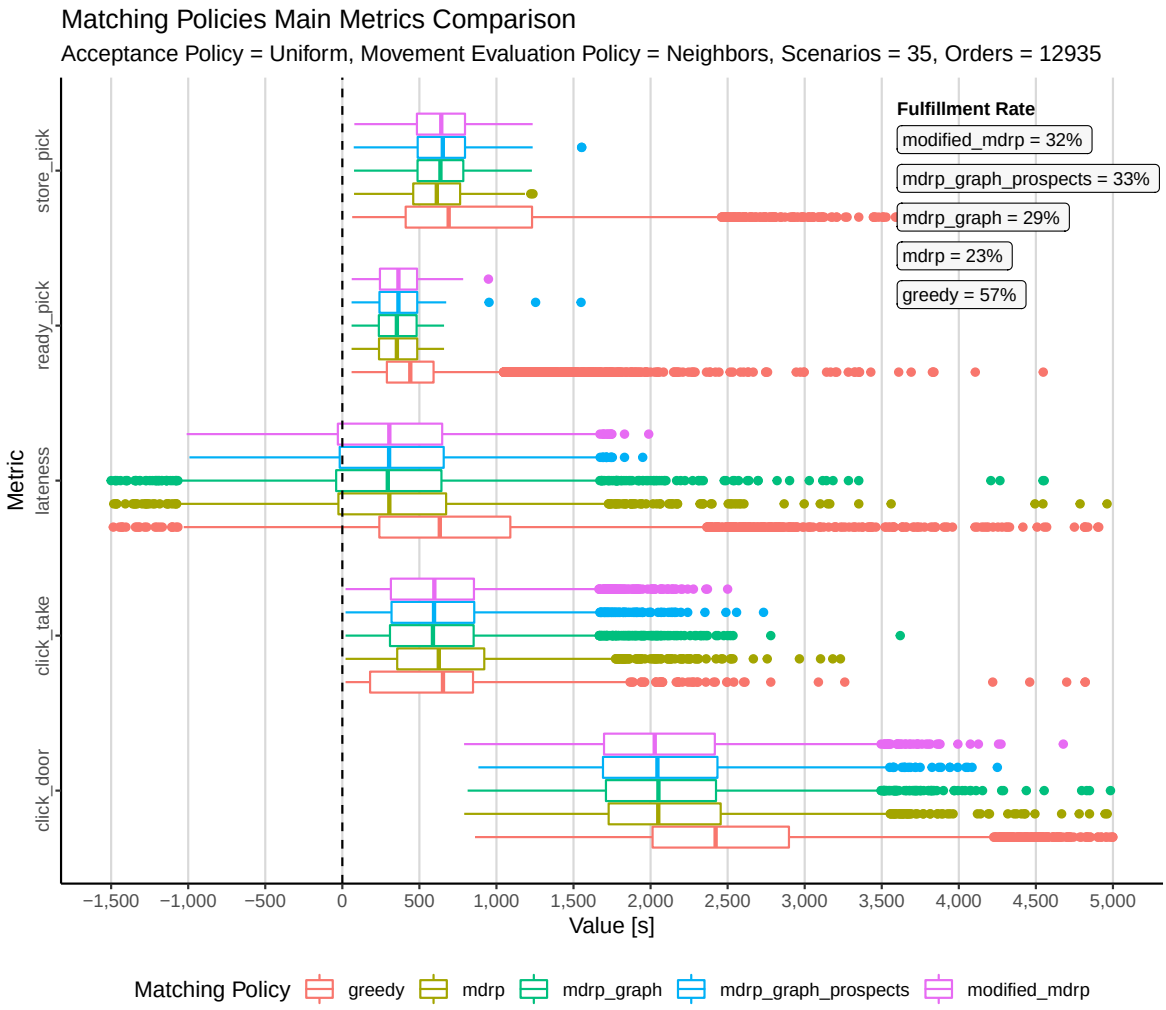


Figure 9: Performance metrics box plots comparison between matching policies for experiment B (uniform-neighbors) across scenarios.

ready time. A more robust modification is to probe possible matches before the order is notified to the restaurant, thus avoiding food waste if no courier will be available to match with the order.

6.3.3 Inspection Across Scenarios

After analyzing the performance of the five *matching* policies across 36 scenarios under two operational configurations, a further inspection of the `mdrp` policy for Experiment B (`uniform-neighbors`) is conducted to more precisely understand how this policy varies its behavior under different conditions of the demand and fleet availability in a more realistic setting, given it shows the inferior fulfillment rate. Figures 10 and 11 show how the `mdrp` policy performs across the scenarios described in Appendix B under Experiment B. A vertical line crosses zero for reference.

The `mdrp` *matching* policy performs consistently for scenarios 13 - 31, as seen in the click to door time. The inter-quartile ranges oscillate around 1500 to 2500 seconds for these scenarios, with small variations in the median and no outliers. These scenarios contain small and medium instances with varied fleet conditions. The behavior is less consistent for scenarios 1 - 12, corresponding to the large instances with a limited fleet. Although the inter-quartile ranges oscillate between the same range, the median has a higher variability. Outliers are present for scenarios 1 - 12, meaning city-specific conditions should be examined. The worst click to door time is observed for scenarios 33 - 36, belonging to medium instances over a demand peak with a limited fleet. For this stringent conditions, the *matching* policy causes higher delivery times over an adjusted variability, surpassing the 41 minute mark (2500 seconds). For a meal delivery operation, having consistent deliveries over 40 minutes is not a desired behavior.

The click to taken time behaves similarly to the click to door time: consistent results for scenarios 13 - 31, increased times for scenarios 33 - 36 and unpredictable distributions for scenarios 1 - 12. The higher click to taken time for scenarios 33 - 36 suggests problems with the fleet distribution, as it takes longer for an order to be assigned, due to either lack of couriers or a high rejection rate.

The drop-off lateness performance of the `mdrp` *matching* policy is adequate in scenarios 13 - 31, since the distribution contains the reference, hinting at an on-time operation. For some deliveries, arriving before the expected drop-off time is ideal, although this must be handled with care as some customers have reservations over their meals arriving before the expected time, since they may not be at the delivery address yet. Given that for scenarios 33 - 36 the click to taken time is higher, it is shown in the drop-off lateness time, given that more deliveries are fulfilled after the expected drop-off time. The `mdrp` policy's performance is sub-optimal for the large instances, as there is uncertainty evidenced for the drop-off lateness metric for scenarios 1 - 12.

The ready to pick-up time is consistent across scenarios, with a median around 300 seconds for most of them. Most scenarios show maximum values slightly above the 10-minute mark, which is the limit for an expected loss of freshness. However, around 25% of the orders evidence a ready to pick-up time of around 200 seconds which is a positive outcome.

The in-store to pick-up time describes the quality of the matches for the couriers. It can be seen that for scenarios 33 - 36 there are smaller medians when compared to scenarios 13 - 31, meaning that the couriers spend less time at the restaurant. Contrasting against the click to taken time leads to the conclusion that the `mdrp` policy causes late arrivals to the restaurant, affecting the freshness of the meals but benefiting the couriers. This shows the real-life situation in which stakeholders have conflictive interests.

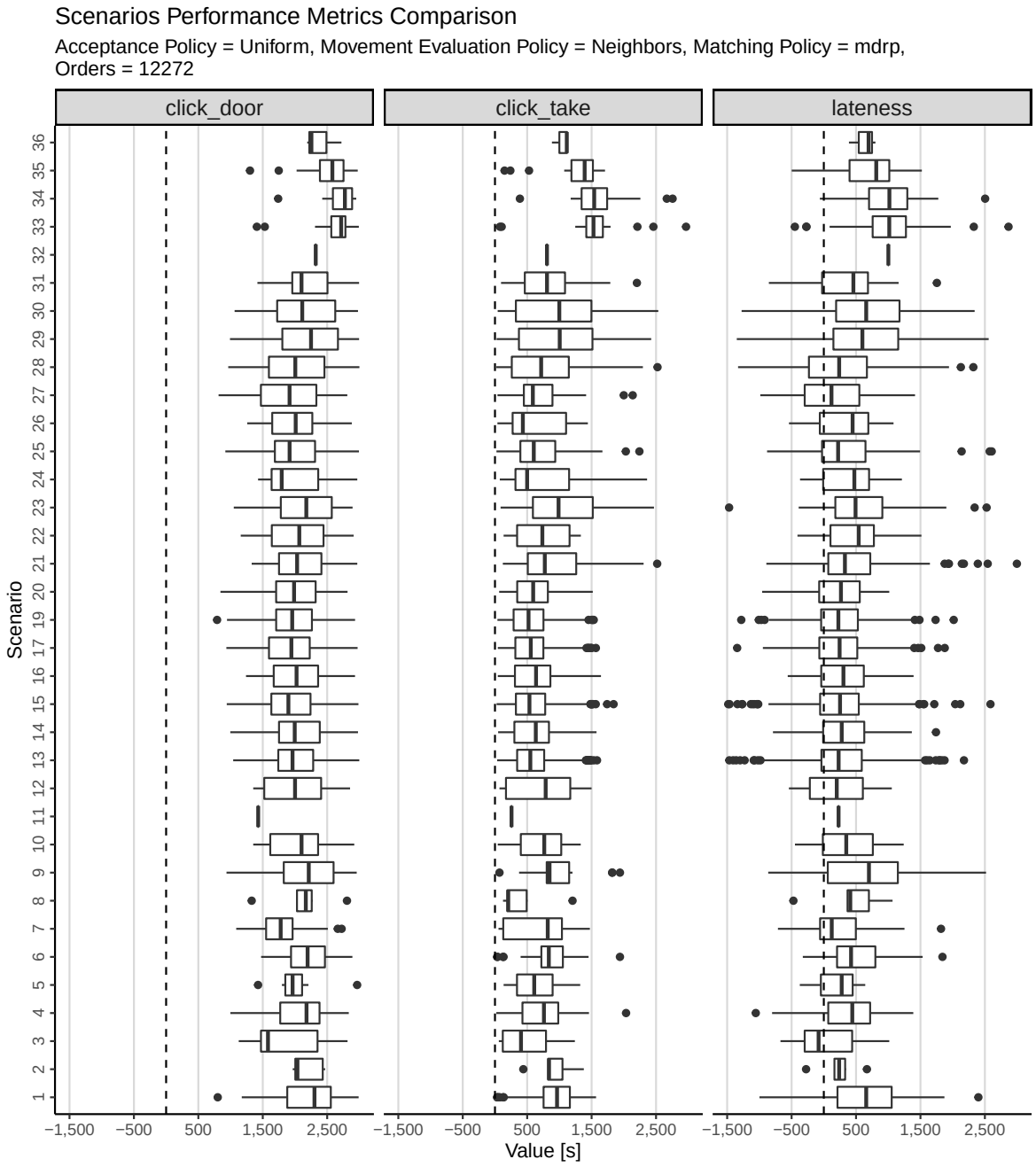


Figure 10: Performance metrics box plots for the uniform-neighbors experiment (B) and mdrp matching policy. Comparison across scenarios. Metrics: click to door time, click to taken time, drop-off lateness time.

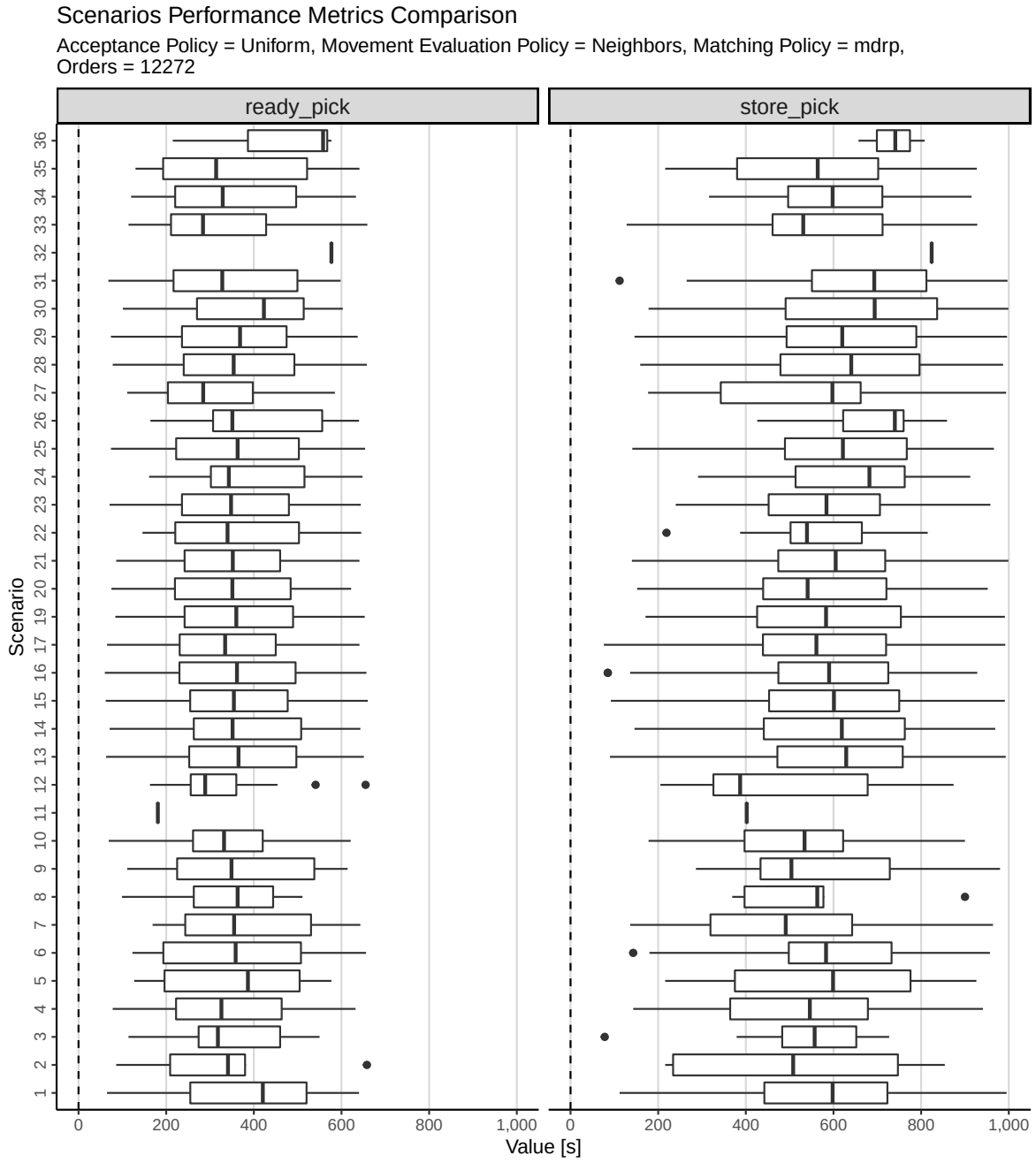


Figure 11: Performance metrics box plots for the uniform-neighbors experiment (B) and mdrp matching policy. Comparison across scenarios. Metrics: ready to pick-up time, in-store to pick-up time.

6.3.4 Computational Efficiency

To understand how the *matching* policies compare regarding computational efficiency, Figure 12 compares them by computational time box plots, differentiating algorithms (matching and routing).

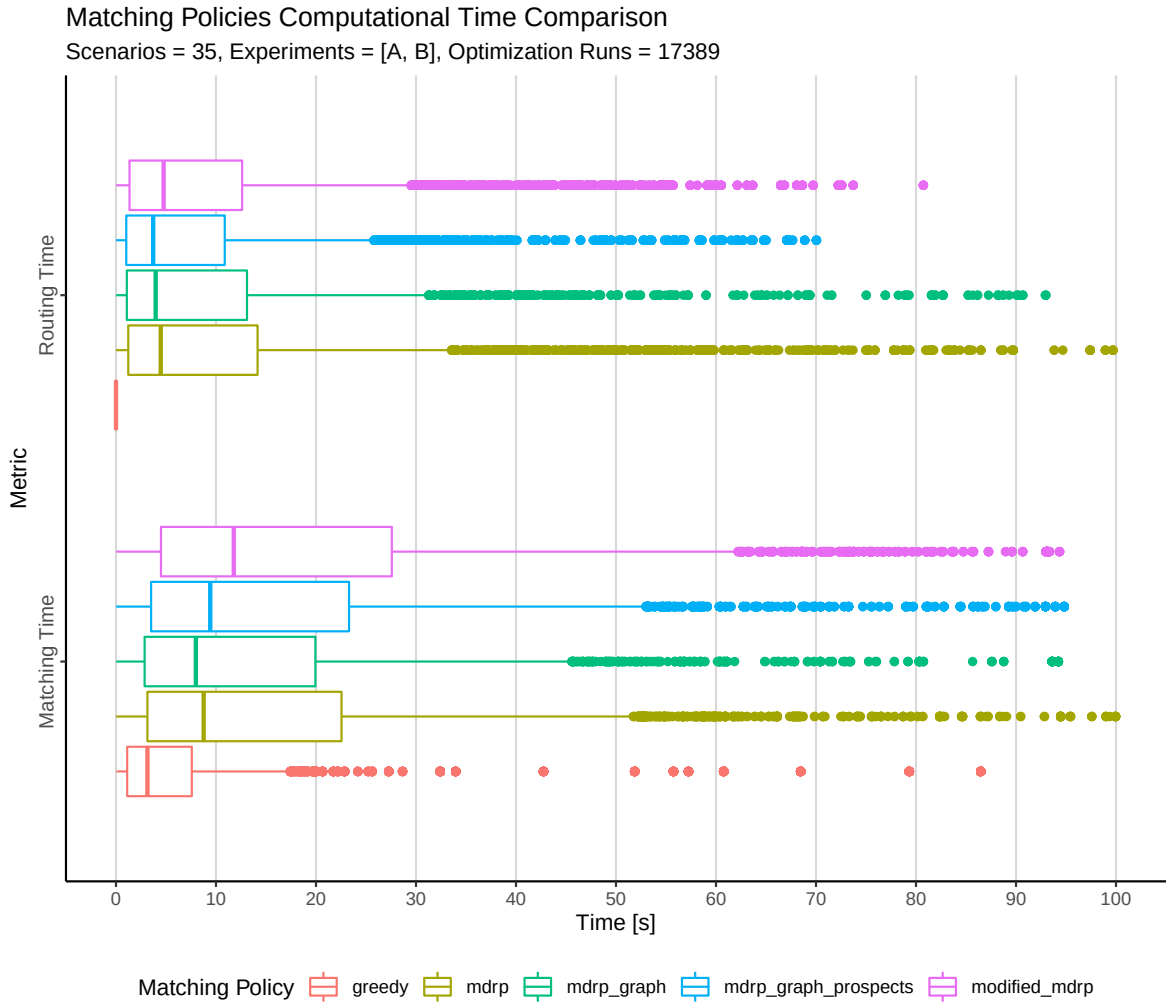


Figure 12: Computational time box plots comparison between matching policies. Algorithm times shown: matching and routing.

The `greedy` *matching* policy converts every order into a single route and does not register routing times. The `mdrp`, `mdrp-graph` and `mdrp-graph-prospects` *matching* policies evidence similar routing time performance as a consequence of using Algorithm 3 without assignment updates. The `modified-mdrp` *matching* policy has a slightly higher median, hinting at assignment updates affecting routing efficiency caused by the additional computations. The distributions of the four optimization *matching* policies are alike, all showing a high quantity of outliers.

The `greedy` *matching* policy shows the best matching computational performance due to its simple logic. In contrast, the median for the matching time in the other *matching* policies is three to four times higher. The `mdrp-graph` *matching* policy shows the best computational performance with the most adjusted distribution and lower values, with the `mdrp` policy close behind. This shows that implementing the network flow model from Definition 5.11 (used in the `mdrp-graph`, `mdrp-graph-prospects` and `modified-mdrp`) is not decisive when compared to the MIP model from Definition 5.10, corresponding to the `mdrp` policy.

The prospects strategy described in Definition 5.12 is conceived to reduce the solution space of the matching problem and gain computational efficiency, however, it is shown that the overhead of calculating the prospects negatively impacts the overall matching time and is not advantageous in this regard, evidencing slower times for the `modified-mdrp` policy. The computational times for the routing and matching algorithms are very high when compared to expected results from a production-ready model, which is typically expected to compute problems of similar magnitude in few seconds (milliseconds when possible). From a research perspective comparing *matching* policies is beneficial, however alternative policies should be considered for a production solution, aiming at using modern distributed and scalable cloud computing. In many production *matching* policies, some sort of problem partitioning strategy is used, such as clustering, to solve smaller models in parallel.

The optimization-based *matching* policies use different strategies for variable creation and embed distinct models. To compare these four policies, Figure 13 depicts the solution spaces, with respect to variables and constraints, for optimization runs across scenarios and experiments (A and B).

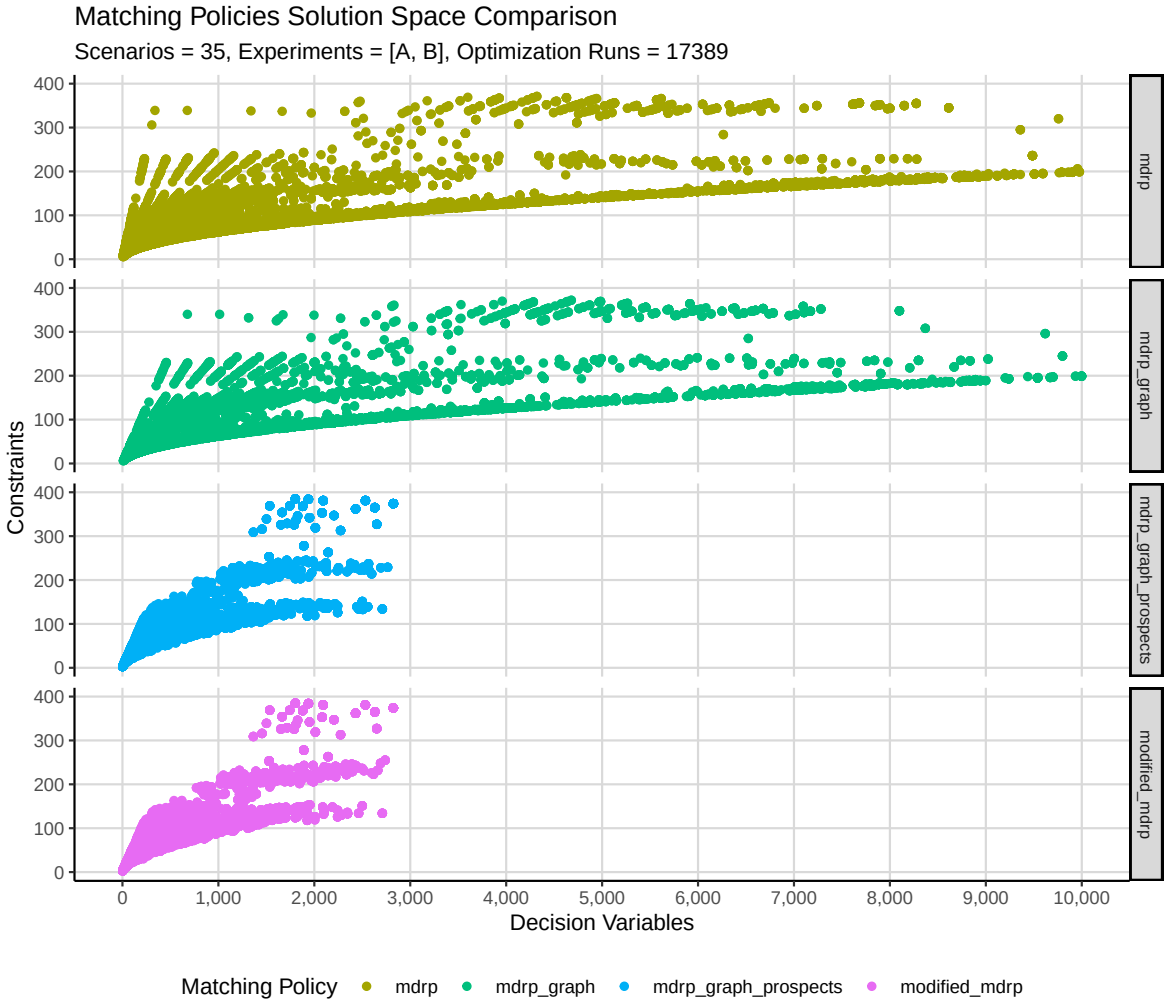


Figure 13: Solution space comparison between optimization matching policies, across scenarios and experiments.

The prospects strategy limits the creation of decision variables which directly results in fewer constraints. As seen from Figure 12, the `mdrp` and `mdrp-graph` policies show the best computational time, meaning that removing low quality bases with the prospects strategy does not lead to a better search of the solution space. The pre-solve stage of the solver takes care of removing sub-optimal solu-

tions in the `mdrp` and `mdrp-graph` policies. In this research, the prospects strategy does not evidence an advantage from the computational efficiency perspective. If the problem turns prohibitively large, the prospects strategy may result beneficial. Additional runs should be performed with the `COIN-OR` engine to compare results among solvers.

7 Conclusions

A flexible computational framework for solving the Meal Delivery Routing Problem is introduced. The framework leverages on interchangeable policies to test out models and algorithms that solve different challenges of the MDRP. It is designed to easily load real-life instances, simulate and output performance metrics for comparison. The framework can be easily plugged in to any data source and leverages the use of external services for tasks such as optimization and city routing. The proposed instances allow for a variety of operational configurations where hypotheses and solutions may be tested. In addition, new definitions are given and formalized in the MDRP to resemble a real operation.

Computational experiments were conducted using the framework over 36 scenarios, comparing two operational conditions and five *matching* policies. It is shown that for both less realistic (couriers who stay still and accept any notification) and realistic (couriers move about the city and decide whether to accept a notification) operations, the `modified-mdrp` or `mdrp-graph-prospects` policies perform the best since click-to-door, in-store to pick-up, drop-off lateness, click to taken and ready to pick-up times are minimized and have a more adjusted distribution. The `greedy` has the best overall fulfillment rate but with sub-optimal quality regarding times. This situation resembles a life-like business problem: different models optimize separate metrics and thus a decision must be made over which metric has priority. When dissecting the performance of the `mdrp` policy in the realistic operation setting, the model under-performs when the service quality is more stringent and the system is less tolerant to delays.

The novel definitions allow for an abstraction of the meal delivery problem into a more generalized On-Demand Delivery Routing Problem. Given that an order has a placement, preparation and ready times and there are service times associated to the store and the user, the problem can be easily transformed into a Groceries Delivery Routing Problem. Orders may be scheduled well ahead of time, in which case the placement time is much less than the preparation time. Routing passengers for the Dial-A-Ride Problem can also be abstracted using these definitions. An order (ride) has a pick-up location (the user's location), a drop-off location (the desired destination), a placement time (now), a preparation time (now or scheduled), a ready time (the same as the preparation time) and service times (which are zero at both locations). The problem is the same: finding a set of routes that optimize some performance metric by matching couriers (drivers) to orders (riders). It can be seen that many on-demand problems may be abstracted to a single definition.

Given that the foundational work for representing the operation has been done, further research can focus on developing new policies. The policies showcased in this research are fairly simple and have a myopic approach. New work with look-ahead policies can be done to proactively preposition couriers about the city, based on estimations of the demand. *Cancellation* policies can be modified to incorporate the variability in preparation times and develop robust models where an order can be canceled before starting the preparation if no courier is estimated to be available for matching. Additionally, different routing approaches can be compared while also considering assignment updates. A single cost function was used to produce the matches, but different metrics can be optimized to compare outcomes. Lastly, a more robust definition can be given to the simulator, by defining new actors such as the restaurant and real-life decisions that impact the operation.

A Glossary

a_o: placement time of the order $o \in O$

β: maximum allowed total off-set from the stops' expected time

b_{ℓ_1,ℓ_2}: travel time between locations ℓ_1 and ℓ_2

C: set of couriers

c: a courier $c \in C$

c_n: the courier $c \in C$ associated to notification $n \in n$

c_o: the courier $c \in C$ assigned to order $o \in O$

C_r: set of couriers available at restaurant $r \in R$

C_t: idle and picking-up couriers at time $t \in T$

d: maximum linear distance between a courier and a restaurant

d_o: preparation time of order $o \in O$

$\delta^s_{o,c}$: drop-off time of order $o \in O$ in route $s \in S$ if assigned to courier $c \in C$

Δ_1: maximum time after an order has been ready

e_c: on-time of courier $c \in C$

e_o: ready time of the order $o \in O$

f: buffering time for sending pick-up & drop-off notifications

f_o: actual drop-off time for order $o \in O$

f'_o: expected drop-off time for order $o \in O$

f_p: time fraction for sending prepositioning notifications

f_r:time fraction for courier relocation

$g_{c,s}$: assignment cost if courier $c \in C$ where to be matched to route $s \in S$

h_{ℓ_1,ℓ_2}: haversine distance between locations ℓ_1 and ℓ_2

I: set of inputs for a policy

i_s: insertion position at route $s \in S$

l_c: off-time of courier $c \in C$

ℓ': location of a city intersection

ℓ_c: on-location of courier $c \in C$

$\ell_{c,t}$: location of courier $c \in C$ at time $t \in T$

ℓ_n: location of notification $n \in N_t$

ℓ_r: pick-up location for restaurant $r \in R$

ℓ_u: drop-off location of user $u \in U$

m_c: number of orders delivered during courier's $c \in C$ shift

$MDRP$: Meal Delivery Routing Problem

m_r: number of routes at restaurant $r \in R$

N_t: set of notifications at time $t \in T$

n: a notification $n \in N_t$

n_p: a prepositioning notification $n_p \in N_t$

n_{pd}: a pick-up & drop-off notification $n_{pd} \in N_t$

O: set of orders

o: an order $o \in O$

O_t: unassigned orders at time $t \in T$

$\mathbb{P}$: a policy

$\mathbb{P}'[\ldots]$: a particular policy

P': output of a policy

p_1: fixed compensation per order delivered

p_2: fixed compensation rate per hour

P_c: acceptance probability of courier $c \in C$

p_c: acceptance rate of courier $c \in C$

$\pi_{s,c}$: pick-up time of route $s \in S$ if assigned to courier $c \in C$

$\mathbb{P}_{ap}$: courier acceptance policy

$\mathbb{P}_b$: dispatcher buffering policy

$\mathbb{P}_{dc}$: dispatcher cancellation policy

$\mathbb{P}_{dp}$: dispatcher prepositioning policy

$\mathbb{P}_m$: dispatcher matching policy

$\mathbb{P}_{me}$: courier movement evaluation policy

$\mathbb{P}_{mp}$: courier movement policy

$\mathbb{P}_{pe}$: dispatcher prepositioning evaluation policy

$\mathbb{P}_{uc}$: user cancellation policy

R: set of restaurants

r: a restaurant $r \in R$

r_o: restaurant of the order $o \in O$

S: set of routes

s: a route $s \in S$

s_c: the current route $s \in S$ of courier $c \in C$

S_{max}: maximum number of orders per route

s_n: the route $s \in S$ associated to notification $n \in n$

S_r: set of routes for restaurant $r \in R$

s^r: pick-up service time at restaurant $r \in R$

S_t: set of routes calculated at time $t \in T$

s^u: drop-off service time at user $u \in U$

T: time horizon

t: moment in the time horizon $t \in T$

τ: system target drop-off time

t^d: cancellation time for the dispatcher

θ: penalty for a delayed drop-off

t^u: cancellation time for user $u \in U$

U: set of users

u: a user $u \in U$

u_o: user of the order $o \in O$

V: set of vehicles

v: a vehicle $v \in V$

v_c: vehicle $v \in V$ of courier $c \in C$

$\emptyset$: empty set

X: a random variable

$x_{c,s}$: variable in assignment models

x_u: cancellation probability of user $u \in U$

Z_t: target route size at time $t \in T$

B Complete Scenarios Description

The following policies are used across all scenarios:

- User *cancellation* policy ($\mathbb{P}_{uc}$) $\implies$ $\mathbb{P}'_{uc}$[random]
- Dispatcher *cancellation* policy ($\mathbb{P}_{dc}$) $\implies$ $\mathbb{P}'_{dc}$[static]
- Dispatcher *buffering* policy ($\mathbb{P}_{b}$) $\implies$ $\mathbb{P}'_{b}$[rolling-horizon]
- Dispatcher *prepositioning evaluation* policy ($\mathbb{P}_{pe}$) $\implies$ $\mathbb{P}'_{pe}$[fixed]
- Dispatcher *prepositioning* policy ($\mathbb{P}_{dp}$) $\implies$ $\mathbb{P}'_{dp}$[naive]
- Courier *movement* policy ($\mathbb{P}_{mp}$) $\implies$ $\mathbb{P}'_{mp}$[osrm]

The following values are used across all scenarios:

- Courier acceptance rate (p_c) = $U(0.4, 1)$
- Courier movement probability = 0.4
- Courier wait to move (f_r) = 45 minutes
- Courier earnings per order (p_1) = 3
- Courier earnings per hour (p_2) = 8
- Dispatcher rolling horizon time (f) = 2 minutes
- Dispatcher prepositioning time (f_p) = 1 hour
- Dispatcher prospects maximum distance (d) = 3 km
- Dispatcher maximum orders per route (S_{max}) = 3
- Dispatcher prospects maximum off-set (β) = 10 minutes
- Dispatcher prospects maximum ready time (part of the commitment strategy) = 4 minutes
- Dispatcher maximum ready time slack (Δ_1) = 10 minutes
- Dispatcher delay penalty (θ) = 0.4
- User cancellation probability (x_u) = 0.75
- User service time (s^u) = $U(2, 5)$ minutes
- Restaurant service time (s^r) = $U(2, 10)$ minutes
- Order target drop-off time (τ) = 40 minutes

All configurations can be modified in the `settings.py` file of the repository (Quintero, 2020).

Scenarios are configured based on the different instances. In addition, the simulation's *start time* (ST), *end time* (ET), *create users from* (CUF), *create users until* (CUU), *create couriers from* (CCF), *create couriers until* (CCU) and *warm-up time* (W - in seconds) are varied. The other two variations among scenarios are the dispatcher's cancellation time (t^d, in minutes) and the user's cancellation time (t^u, in minutes). The simulation scenarios are listed in Table 3 based on different configurations of these 10 variations. In Table 3, there are different groups of scenarios:

1. *1 - 12.* Scenarios over the large instances, with a limited fleet, small window of order creation and a strict cancellation behavior (both the dispatcher and the user don't want to wait before deciding to cancel).

2. *13 - 20*. Scenarios over the small instances, with a large fleet, a whole day of order creation and a strict cancellation behavior.
3. *21 - 28*. Scenarios over the small instances, with a medium fleet, order creation based on the valley hours and a tolerant cancellation behavior (the dispatcher and user wait more time before deciding to cancel).
4. *29 - 32*. Scenarios over the medium instances, with a limited fleet, order creation based on the valley hours and a tolerant cancellation behavior.
5. *33 - 36*. Scenarios over the medium instances, with a limited fleet, small window of order creation based on the nightly peak hours and a permissive cancellation behavior.

Table 3: Configuration of scenarios

Scenario	**Instance**	**ST**	**ET**	**CUF**	**CUU**	**CCF**	**CCU**	**W**	t^d	t^u
1	0	10:00	18:00	16:00	16:05	10:00	10:10	18 000	25	15
2	1	10:00	18:00	16:00	16:05	10:00	10:10	18 000	25	15
3	2	10:00	18:00	16:00	16:05	10:00	10:10	18 000	25	15
4	6	10:00	18:00	16:00	16:05	10:00	10:10	18 000	25	15
5	7	10:00	18:00	16:00	16:05	10:00	10:10	18 000	25	15
6	8	10:00	18:00	16:00	16:05	10:00	10:10	18 000	25	15
7	12	10:00	18:00	16:00	16:05	10:00	10:10	18 000	25	15
8	13	10:00	18:00	16:00	16:05	10:00	10:10	18 000	25	15
9	14	10:00	18:00	16:00	16:05	10:00	10:10	18 000	25	15
10	18	10:00	18:00	16:00	16:05	10:00	10:10	18 000	25	15
11	19	10:00	18:00	16:00	16:05	10:00	10:10	18 000	25	15
12	20	10:00	18:00	16:00	16:05	10:00	10:10	18 000	25	15
13	3	00:00	23:59	00:00	22:00	00:00	06:00	10 800	25	15
14	4	00:00	23:59	00:00	22:00	00:00	06:00	10 800	25	15
15	9	00:00	23:59	00:00	22:00	00:00	06:00	10 800	25	15
16	10	00:00	23:59	00:00	22:00	00:00	06:00	10 800	25	15
17	15	00:00	23:59	00:00	22:00	00:00	06:00	10 800	25	15
18	16	00:00	23:59	00:00	22:00	00:00	06:00	10 800	25	15
19	21	00:00	23:59	00:00	22:00	00:00	06:00	10 800	25	15
20	22	00:00	23:59	00:00	22:00	00:00	06:00	10 800	25	15
21	3	07:00	12:00	07:00	11:00	07:00	09:00	3 600	60	45
22	4	07:00	12:00	07:00	11:00	07:00	09:00	3 600	60	45
23	9	07:00	12:00	07:00	11:00	07:00	09:00	3 600	60	45
24	10	07:00	12:00	07:00	11:00	07:00	09:00	3 600	60	45
25	15	07:00	12:00	07:00	11:00	07:00	09:00	3 600	60	45
26	16	07:00	12:00	07:00	11:00	07:00	09:00	3 600	60	45
27	21	07:00	12:00	07:00	11:00	07:00	09:00	3 600	60	45
28	23	07:00	12:00	07:00	11:00	07:00	09:00	3 600	60	45
29	5	00:00	10:00	00:00	08:00	00:00	00:10	8 400	60	45
30	11	00:00	10:00	00:00	08:00	00:00	00:10	8 400	60	45
31	17	00:00	10:00	00:00	08:00	00:00	00:10	8 400	60	45
32	22	00:00	10:00	00:00	08:00	00:00	00:10	8 400	60	45
33	5	08:00	21:00	19:00	19:05	08:00	08:30	18 000	50	30
34	11	08:00	21:00	19:00	19:05	08:00	08:30	18 000	50	30
35	17	08:00	21:00	19:00	19:05	08:00	08:30	18 000	50	30
36	22	08:00	21:00	19:00	19:05	08:00	08:30	18 000	50	30

C LINPACK Benchmark

This Appendix contains the steps necessary to benchmark an **Intel-based** CPU.

1. Go to the Intel® Math Kernel Library (Intel® MKL) Benchmarks Suite.
2. Download the latest package that is appropriate for the operating system (for this research, a macOS was used).
3. Unzip the file.
4. Go to this dir (mac version for 2020 is shown): `./benchmarks_2020/mac/mkl/benchmarks/linpack`
5. Double click the `linpack_cd64` (or appropriate file) Unix executable.
6. A terminal/command prompt pops up, press Enter.
7. For the `Number of equations to solve` enter 5000 (or a large enough number).
8. For the `Leading dimensions for array` enter 10000 (or a large enough number).
9. For the `Number of trials to run` enter 200 (or a similar number).
10. For the `Data alignment value (in Kbytes)`, which is the memory reservation to use, enter 1000000 (or the desired memory allocation).
11. The program starts running and showing benchmarks.
12. When finished, use the `Average` GFlops result reported as the benchmark.

References

Berbeglia, G., Cordeau, J.-F., & Laporte, G. (2010). Dynamic pickup and delivery problems. *European Journal of Operational Research*, *202*(1), 8–15. https://doi.org/https://doi.org/10.1016/j.ejor.2009.04.024

Castillo, J., Knoepfle, D., & Weyl, G. (2017). Surge pricing solves the wild goose chase, 241–242. https://doi.org/10.1145/3033274.3085098

COIN OR Foundation. (2020). *Coin-or: Open source for the operations research community*. COIN-OR Foundation. https://www.coin-or.org/

Cordeau, J.-F., & Laporte, G. (2007). The dial-a-ride problem (darp): Models and algorithms. *Annals OR*, *153*, 29–46. https://doi.org/10.1007/s10479-007-0170-8

Docker Inc. (2020). *Docker*. Docker Inc. https://www.docker.com/

Dongarra, J. J., Luszczek, P., & Petitet, A. (2003). The linpack benchmark: Past, present and future. *Concurrency and Computation: Practice and Experience*, *15*(9), 803–820. https://doi.org/https://doi.org/10.1002/cpe.728

Forbes. (2020). *El servicio a domicilio se volvió fundamental tras el coronavirus*. Retrieved November 23, 2020, from https://forbes.co/2020/05/29/tecnologia/el-servicio-a-domicilio-se-volvio-fundamental-tras-el-coronavirus/

Gurobi Optimization, LLC. (2020). *Gurobi*. Gurobi Optimization, LLC. https://www.gurobi.com/

Hildebrandt, F., & Ulmer, M. (2020). Supervised learning for arrival time estimations in restaurant meal delivery.

Klapp, M. A., Erera, A. L., & Toriello, A. (2018). The dynamic dispatch waves problem for same-day delivery. *European Journal of Operational Research*, *271*(2), 519–534. https://doi.org/https://doi.org/10.1016/j.ejor.2018.05.032

Klein, D. (2019). *Fighting for share in the 16.6 billion delivery app market*. Retrieved November 23, 2020, from https://www.qsrmagazine.com/technology/fighting-share-166-billion-delivery-app-market

Kumar, H. (2020). *Key on-demand app statistics 2020: Survey, research, and figures*. Retrieved November 23, 2020, from https://www.techtic.com/blog/on-demand-app-statistics/

Liao, W., Zhang, L., & Wei, Z. (2020). Multi-objective green meal delivery routing problem based on a two-stage solution strategy. *Journal of Cleaner Production*, *258*, 120627. https://doi.org/https://doi.org/10.1016/j.jclepro.2020.120627

Liu, Y. (2019). An optimization-driven dynamic vehicle routing algorithm for on-demand meal delivery using drones. *Computers and Operations Research*, *111*, 1–20. https://doi.org/https://doi.org/10.1016/j.cor.2019.05.024

Mitrović-Minić, S., & Laporte, G. (2004). Waiting strategies for the dynamic pickup and delivery problem with time windows. *Transportation Research Part B: Methodological*, *38*(7), 635–655. https://doi.org/https://doi.org/10.1016/j.trb.2003.09.002

OSRM. (2020). *Osrm: Open source routing machine*. OSRM. http://project-osrm.org/

Pillac, V., Gendreau, M., Guéret, C., & Medaglia, A. (2013). A review of dynamic vehicle routing problems. *European Journal of Operational Research*, *225*, 1–11. https://doi.org/10.1016/j.ejor.2012.08.015

Pillac, V., Guéret, C., & Medaglia, A. L. (2012). An event-driven optimization framework for dynamic vehicle routing. *Decision Support Systems*, *54*, 414–423. https://doi.org/10.1016/j.dss.2012.06.07

Psaraftis, H. N., Wen, M., & Kontovas, C. A. (2016). Dynamic vehicle routing problems: Three decades and counting. *Networks*, *67*(1), 3–31. https://doi.org/https://doi.org/10.1002/net.21628

Python Software Foundation. (2020). *Python*. Python Software Foundation. https://www.python.org/

Quintero, S. (2020). *Mdrp-sim: The meal delivery routing problem simulator*. https://github.com/sebastian-quintero/mdrp-sim

Reyes, D., Erera, A., Savelsbergh, M., Sahasrabudhe, S., & O'Neil, R. J. (2018). The meal delivery routing problem. *Optimization Online*.

Steever, Z., Karwan, M., & Murray, C. (2019). Dynamic courier routing for a food delivery service. *Computers and Operations Research*, *107*, 173–188. https://doi.org/https://doi.org/10.1016/j.cor.2019.03.008

Team SimPy. (2020). *Simpy: Discrete event simulation for python*. Team SimPy. https://simpy.readthedocs.io/en/latest/

The PostgreSQL Global Development Group. (2020). *Postgresql*. The PostgreSQL Global Development Group. https://www.postgresql.org/

Ulmer, M., Thomas, B., Campbell, A., & Woyak, N. (2017). The restaurant meal delivery problem: Dynamic pick-up and delivery with deadlines and random ready times. *Transportation Science*. https://doi.org/10.1287/trsc.2020.1000

Voccia, S., Campbell, A., & Thomas, B. (2015). The same-day delivery problem for online purchases. https://doi.org/10.13140/RG.2.1.3762.7606

Yan, C., Zhu, H., Korolko, N., & Woodard, D. (2019). Dynamic pricing and matching in ride-hailing platforms. *Naval Research Logistics (NRL)*. https://doi.org/10.1002/nav.21872

Yeo, L. (2020). *Which company is winning the restaurant food delivery war?* Retrieved November 23, 2020, from https://secondmeasure.com/datapoints/food-delivery-services-grubhub-uber-eats-doordash-postmates/

Yildiz, B., & Savelsbergh, M. (2019). Provably high-quality solutions for the meal delivery routing problem. *Transportation Science*, *53*(5), 1372–1388. https://doi.org/10.1287/trsc.2018.0887