An Innovative Last Mile Logistics based on Hybrid Subway Deliveries in Urban Areas

Federico Galbiati

Liceo Scientifico D. Bramante, 20013 Magenta (MI), Italy

Abstract

E-commerce growth is expected to reach 4.8 trillion USD by 2021 and statistics show that 60% of Europeans have purchased online in the past 12 months. A research in 2018 discovered that 54% of US consumers under the age of 25 consider same-day shipping their number one purchase driver. The e-commerce final mile is complex and resource-intense and this research tried to find a reliable and inexpensive solution to this phase of the delivery process. This project suggests the use of an innovative logistics system for last-mile deliveries based on currently existing subway infrastructures. Metro systems can transport packages from warehouses outside a city to subway stations in any district. Once arrived, small vehicles such as cars, bikes, or scooters can be used to deliver to the customer. This strategy can allow building a capillary network for fast deliveries, enabling same day, one hour, and potentially even 30 minutes deliveries. The proposed logistics process was modeled with several computational algorithms to generate and optimize the logistics network. This was done with the development of custom Computer Science algorithms. The most appropriate stations in a city were detected to maximize delivery efficiency. Minimum travel paths were computed with multiple algorithms, real-time genetic modeling, and graph theory. Hybrid subway deliveries were found able to transfer up to one million daily packages (mix of cardboard boxes and envelopes) per line and fulfill a 1 h delivery requirement. The hybrid subway delivery process was found to reduce CO_2 emission by four times compared to standard ground deliveries and increase the drivers' availability by an average of 40%.

Introduction

The e-commerce distribution process in urban areas is, together with private vehicles, one of the major sources of energy consumption, noxious gas emissions and noise levels. This results in the prominent negative life impacts and environmental issues of our cities. Moreover, the solution to an efficient one-hour delivery process is yet to be found. Today, e-commerce makes couriers and especially the last mile fundamental [17]. Based on multiple industry surveys [3, 4, 17], 54% of people interviewed would give repeat business to a retailer with delivery tracking, and 53% of shoppers would not buy an object without knowing a delivery timeline. A scientific literature review was conducted and several innovative delivery concepts were found. Studies have been conducted to efficiently transport packages over medium and long distances utilizing railways. For example, a research published on the EU Transport Research and Innovation Monitoring and Information System [10] suggested that freight transportation could be used on a global scale for the "first mile" of supply chains. In 2012, a freight train delivered goods on an overnight service for Sainsbury's, the third largest chain of supermarkets in the UK with almost 1500 shops. The shipment went from the Midlands into Euston station, for onward out-of-hours delivery to stores in Central London, using smaller vehicles more suited to the urban environment. TNT performed a similar pilot in 2014 transporting goods by train into Euston station for final delivery by electric vehicles. After the original trial, TNT has been experimenting to understand whether railways can help provide a more reliable service to their customers [12].

Other studies [5, 7, 13] suggest implementing freights as environmental-friendly crowdshipping to pickup/dropoff goods in automated parcel lockers located either inside stations or in other public spaces. Crowdshipping, an innovative delivery model could stimulate better use of currently unexploited transport capacity thus reducing transport costs and emissions. The paper focuses on commuters using the metro rather than the bus since the former is typically more frequent and reliable than the second thus making an additional stop/detour more easily acceptable also for low compensations which are typical for last-mile deliveries [9]. The innovative crowdshipping concept was analyzed as a case study for the city of Rome, Italy. It is important to note that the type of goods to be transported is a critical issue, especially for green crowdshipping. In fact, size and weight can reduce people's availability of acting as crowdshippers, while it is plausible that consumers would be less inclined to receive valuables or personal goods through the crowd using public transportation.

Only one study was found suggesting the potential implementation of subways for express logistics in Shanghai [2]. The group researched that courier transportation is usually affected by overloaded trucks and delays often causing accidents, producing congestions, and generating pollution. The paper suggested implementing a subway rail transit system through the usage of custom subway trains for the transportation of packages. However, this plan can be difficult from a financial perspective and would require extensive changes to the subway stations. Moreover, the research does not take into consideration the possibility of using only some stations as terminals for the unloading of packages.

Hybrid Subway Deliveries

This paper proposes the implementation of an innovative last-mile business-to-consumer process. This model relies on a hybrid delivery system involving urban terminals, subways lines, and couriers to maximize the number of deliveries per hour. Moreover, this can abate the number of vehicles used for goods distribution and reduce the environmental impact. Packages go through logistics warehouses placed at specific subway terminals, and from these stations, they are transported to various locations in the city through all available subway lines.

An algorithm clusters in advance packages based on their target subway station to place them on specific wooden pallets. When the subway train reaches a destination, pallets are unloaded either manually or through automated machines. From the station, packets are delivered to the customers using couriers. An algorithm diverts couriers to each station based on the need. Each courier is assigned a batch of packets and an optimized delivery plan. The travel plan is created using a genetic TSP (Traveling Salesman Problem) algorithm [16, 8]. This allows minimizing their trip duration deliveries to any addresses in the city and in many cases in one hour from the order or less.

This hybrid approach employing the subway and ground delivery operators can reduce by half the distance traveled by surface vehicles to deliver the same quantities of goods. As a consequence, these methods could lead to a reduction of half of the vehicles, such as *Amazon Prime Now* trucks. Moreover, using the subway and smaller ground vehicles such as microcars would prevent the previously cited environmental concerns currently generated by delivery trucks.

1. Physical Infrastructure Design

The main physical infrastructures implied in the creation of this hybrid delivery system are the subway station terminals throughout the city and subway cars with a compartment for keeping packages. The transport chain begins with a courier or company who intends to ship a package.

After dropping off multiple packages at a designated intake subway station, the packets are computed. The software will sort the packages and place them on appropriate pallets based on their final destination. The pallets are then loaded on a designated subway car. The second phase involves the transportation of the pallets to the destination station. The third phase involves the removal of the pallet from the train. Two solutions can be used. An automated system can employ an automated fork, that would minimize removal time. A manual approach would involve operators unloading the pallets manually. The following are the details of the proposed designs and systems.

1.1 Subway Car

To fully understand the process, it is first needed to understand the design of the subway car employed in the transportation of the items to deliver along the underground line. As national standards have to be taken into consideration, this example is based on the dimensions of the *Leonardo* underground train [11]. The width of a single subway car is 2.8 m. The total length is 106.9 m for six cars and the height is approximately 3.5 m. The ground level is at 1.1 m from the bottom. Moreover, the standard European dimensions for pallets (*EPAL*) are 1200 mm × 800 mm × 144 mm. Based on this information, a design that maximizes space usage was generated (Tables 1.1, 1.2, 1.3, 1.4).

Pallets can be unloaded through the two side doors without any modifications to the car structure. Moreover, pallets can be placed on two levels, based on the volume capacity requirements of the subway line. In case of high volume transportation lines, pallets can also be placed in a double row configuration, maximizing the usable space in the subway car. Considering the available space of one subway car, 64 pallets can be fitted, wheres dedicating only half a car, could allow transporting 32 pallets. As the size of the most common medium Amazon packages is 450 mm x 350 mm x 200 mm, the maximum number of elements per pallet is 16 boxes or 360 envelopes. Today, several *PrimeNow* packages are usually envelopes, as customers tend to purchase fewer and smaller items. To place the packages in such a space-efficient manner, conveyor belts on two levels could be used to move the pallets. This would allow pre-pickup positioning to minimize the unloading time.

Table 1.1 - Dimensions of a Leonardo subway car

Subway Car Sizes	Dimension
Length (m)	13
Width (m)	2.8
Door height (m)	2.1
Height (m)	2.4
Conveyor belt height (m)	0.15
Free margins (m)	0.05
Available vertical space (m)	1.42
Maximum pallet vertical space (m)	0.71

Table 1.2 - Amazon boxes and envelopes per pallet

Amazon Package	Amazon Box	Amazon Envelope
Height (m)	0.20	0.04
Width (m)	0.45	0.235
Length (m)	0.35	0.18
Per length of the pallet	2	5
Per width of the pallet	2	4
Per height of the pallet	4	18
Total packages per pallet	**16**	**360**

Table 1.3 - Pallets (EPAL) per subway car

Pallet (EPAL)	Full Car	Half Car
Height	0.14	0.14
Width	1.2	1.2
Length	0.8	0.8
Pallets per subway car length	16	8
Pallets per subway car width	2	2
Pallets per subway car height	2	2
Total pallets per subway car	**64**	**32**

Table 1.4 - Daily transportation volumes

Delivery Details	Most Common Amazon Cardboard Box	Amazon Cardboard Envelope	50% Envelopes and 50% Boxes
Trains per hour	8	8	8
Daily hours of operation	15	15	15
Pallets per train	32	32	32
Packages per pallet	16	360	188
Maximum daily packages volume	**61440**	**1382400**	**721920**

1.2 Automatic pallet loading and unloading infrastructure

The goal of this system is to unload the required pallets in a short time from the subway car. To minimize errors and times, an automated approach appears to be a reliable but expensive solution. Pallets have to be transported from a location (warehouse, street...) to the subway car. A conveyor belt could be used to move them from a street to an elevator and then to the train trough the platform (Figure 1.1).

Figure 1.1 - A Leonardo subway train, side view

A robotic forklift attached in front of the belt could then be used to load or unload the pallets in simply one movement, and therefore in a matter of a couple of seconds after the arrival of the subway train. In case the packages had to be moved to the street level, an elevator could be placed on the other side of the conveyor belt. Overall this system could provide fast and uninterrupted movements of the pallets from a street to the designated subway car. When the subway car arrives, the robotic system can automatically align using standard NFC chips (Near Field Communication) with the precision of a couple of centimeters. Each pallet would be assigned a barcode to know what is being transported and allow tracking at any moment by the customer.

1.3 Manual pallet loading and unloading infrastructure

Using a manual system would require more operators on-site, but allow smaller financial investments. A worker could use a manual or electric forklift to unload the pallet from the subway car. This method would not be automated but could be more reliable and financially sustainable. The unloaded pallets could then be personally transported through an elevator to the subway exit, allowing pickup from couriers.

2. Hybrid Subway Delivery Modelling

The design of this hybrid delivery infrastructure was done using a specific algorithm to optimize the positioning of the subway terminals and to minimize the required physical infrastructure while maximizing the delivery coverage area. To achieve this, an algorithm was developed to analyze a given geographical area and model the logistics network. This algorithm identified the stations on any given subway system that would be most suitable for this hybrid logistics.

The algorithm is divided into various steps and begins with the retrieval of map data for a selected area of interest. As no single data source of subway systems was found, multiple APIs were implemented. Although the *Google Maps API* is the most detailed, access to the subway line information is currently restricted and inaccessible. The *Google API* can therefore only identify the stations in a region of interest, but with no information on the subway lines or sequence of stations. Therefore, another API, *Overpass*, was utilized to pull data of all the subway stations in a geographical area and their coordinates. To parse the subway lines, connecting stations, and routes, an API by *OpenStreet Map (OSM)* was implemented. *OSM* returned YAML content, whereas *Overpass* returned JSON data. Both were parsed and stored in custom *Codable* data structures. As *OSM* and *Overpass* used two identifier styles for the same station, a matching function was created.

After the parsing was completed, the algorithm subdivided the city into rectangular regions of a given size. The algorithm then performed computations for each square region to identify one station among all the ones in the area the would perform best. The criteria was finding a station that would be most centered, minimize the subway lines deployed, and minimize traffic impact. The algorithm performed three main computations. First, it determined which station was the most central based on the coordinates of the locations. Then, the algorithm prioritized stations on the longest subway lines. Trying to concentrate the stations on the most extended ones could lead to fewer transfers between different lines and therefore shorter travel paths. Moreover, *Microsoft Azure* [6] was implemented to compute the most efficient station in terms of traffic. The Traffic Flow Segment endpoint was queried with the coordinates of each station to identify the one with the highest *freeFlowSpeed* value.

After identifying a station for each grid block, the algorithm computed multiple parametric calculations to identify the maximum transport chain volumes and capabilities. These were determined through the dimensions of the train, the temporal frequency of the train schedule, the volume of a pallet (*EPAL* pallets), the volume of a package, and the daily operative hours.

3. Delivery Chain Transport Optimization

Once the optimal logistic model has been computed, a second algorithm is employed to manage the flow of package deliveries from the origin station to the customers. This process involves planning the pallets, finding the nearest unloading stations to the final destinations, and computing the overall scheduling. Moreover, the algorithm determined the ETA of each pallet providing accurate information to the customer about their shipments.

After creating a database of orders, orders can be added to it. Each order contains personal information about the customer, including the address, the building type, weight, and volume. These last three parameters are utilized to compute the delivery time with more accuracy, as unloading and dropping off a box can vary based on its physical characteristics. A *computeTargetStations* function is then executed to identify the target stations for each delivery address. As addresses are not comparable to the coordinates of the stations, Geocoding was implemented. Through the *Google Maps API*, a GET request query function was implemented to determine the coordinates of each given address.

Each address is then identified in the logistic model grid, based on its coordinates. As the grid block station could be slightly off-centered in the block, it could make the destination closer to the stations of adjacent blocks. The 3x3 grid area centered on the block containing a destination is then extracted. The distance between the station of each block and the customer's address is then computed. This is not done through simple coordinate geometry, but by sending a server request to the *Google Maps API*, and computing the path between the two. Moreover, this approach takes into consideration for street traffic.

This algorithm then computed the travel path between the departing terminal and the destination station previously found. This minimum path was computed through a Dijkstra Breadth-First Search based algorithm [1]. The subway lines were analyzed as one graph of stations (vertexes) and edges (station to station connections). The graph edges were weighted such that traversing an edge that implied a train change would have a higher cost.

A data structure was created and contains initially just the source vertex. The adjacent vertexes are then put in a queue based on their distance. Each vertex is visited and this process continues until the destination is found. Each time, the neighboring vertexes are analyzed for their weight. Stations are checked for their previous encounters and each valid adjacent vertex is checked for its distance. If the distance between each adjacent vertex and each current vertex is greater than the minimum distance obtainable with other prioritized nodes, the neighbor is discarded. This calculation relies on the previous steps and the weight of the path to compute correctly.

In the end, a sequence of edges is returned and defines the minimum time for the package to travel from the departing to the arrival stations. The results from this algorithm are passed to part four, responsible for the final delivery logistics between the subway and the customer.

4. Courier Management Algorithm

The last step for the delivery logistics is the delivery of the packages to each address within the expected time frame. To achieve this result, an algorithm was created. The best approach to delivering variable volumes of packages within a fixed time frame is a dynamic short-range delivery system. This short-range system would allow the pickup of packages from the subway station and the delivery to destinations in the square areas as subdivided in part two.

The first step of the algorithm is identifying the number of operators needed to complete the delivery of all requested packages, considering an average delivery time of 5 minutes each, within the designated timeframe.

The second step of the algorithm involves determining the minimum travel path for each delivery operator, in order to deliver a set number of packages from an origin (the unloading subway station) to a set of destinations. This was chosen to be done with a genetic Travelling Salesman Problem (TSP) algorithm. This is preferable to just a minimum travel path because the delivery

man can return to the subway station in the minimum time possible, and will be immediately available for incoming delivery requests. The TSP was provided data from the *Google Maps API*, in order to keep in consideration traffic conditions that could affect delivery ETAs, overall optimization, and optimal transportation flow.

For convenience in accessing the Distance Matrix Services API, the TSP algorithm was run in a mixed JavaScript and Swift environment. A configuration panel was created on the web page to set the population size, mutation and crossover rates, and maximum generations. Generally, the algorithm was run for 50 iterations, as the number of nodes involved in the TSP was relatively small. During unit tests, no better results were obtained increasing the iterations to more than 50.

For each computed destination cluster, the algorithm would return the best travel path obtained. Each path was then verified for its requirements. Results for performing deliveries to each set of destinations were saved and analyzed.

A variable t indicating the maximum time for deliveries to occur in allowed to evaluate each set. When the total delivery time for the set of destinations was greater than t, the algorithm would split the vector of deliveries in half and assign each half to an additional operator, therefore lowering times.

5. Lynx: Custom Data Visualization Web Platform

To visualize the data in a simple and reliable manner, a JSON API was created for the algorithms. The previous algorithms could automatically send their results to a web platform created, called *Lynx*. This web platform was developed in HTML5, CSS3, and JavaScript, using jQuery, Bootstrap for graphics, and the *Google Maps API* to visualize on a map the model created by the algorithm in section 2 and real-time algorithms 3 and 4 (Figure 5.1).

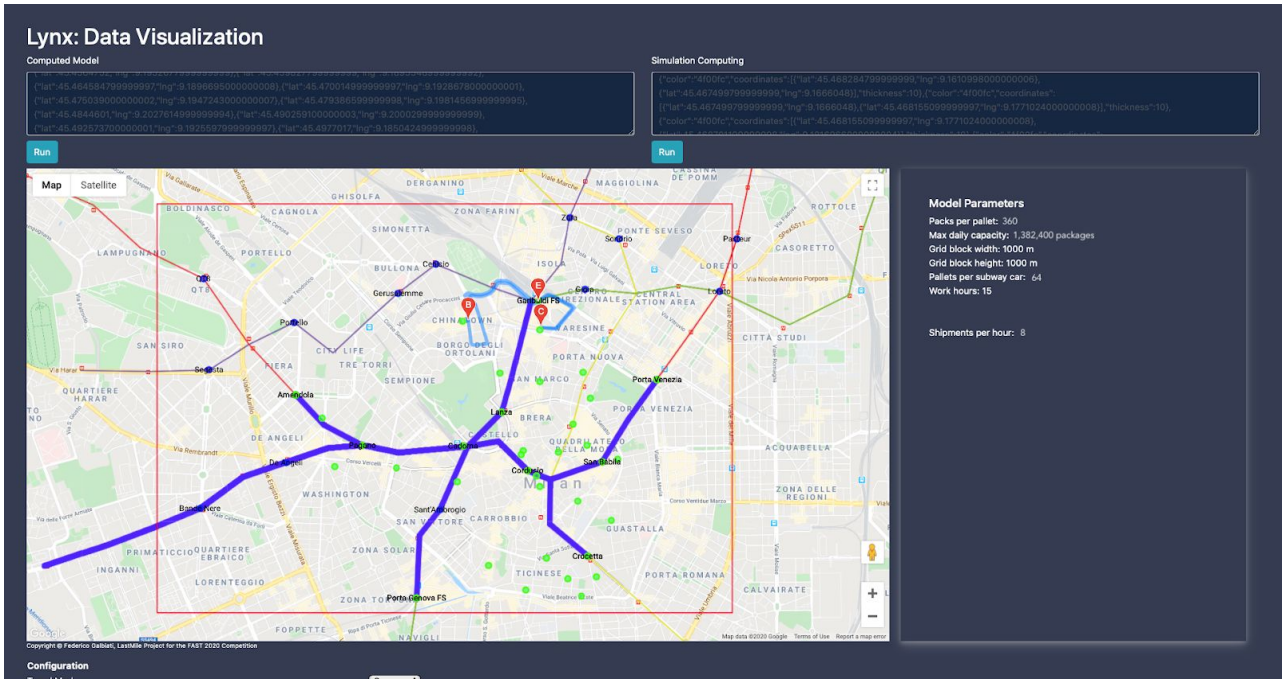

Figure 5.1 - Lynx Data Visualization Platform Screenshot, simulation phase

Custom *Codable* objects were created in Swift and the results of the algorithms previously mentioned were encoded in them. A conversion function encoded the objects in JSON, placing them in real-time on the API, for *Lynx* to fetch.

Subway lines are drawn on the map with the colors defined by the *OSM API,* whereas the simulated travel paths are highlighted in a different and distinguishable color. Markers we set on the map to show the target stations and, in a different color, the destination station.

6. Real-World Simulations And Results

6.1 Milan

A simulation of the Milan metropolitan system returned the following results, using the station of Bisceglie as the departing location. With a 1 km by 1 km grid, the system was modeled to have: 10 stations on the M1, 6 on the M5, 7 on the M2, and 3 on the M3 (Figure 6.2). This coverage allowed 26 randomly chosen destinations to receive deliveries in widely spread areas of Milan within 60 minutes from the dispatching. Each pallet traveled an average of 12-25 minutes on a subway train and then an average of 5 to 10 minutes on a car. The network was also tested positive for performing half an hour deliveries in most of the subway system but requiring an increase in personnel. Computing the same deliveries with ground transportation showed an increase in time to reach the destination zone of about 10 minutes. Moreover, the road distance traveled decreased on average by four times (Table 6.1). Instead, the duration of the round trips computed by the TSP decreased on average by 42% (Table 6.2), indicating an increased operator availability by 30 and 70%.

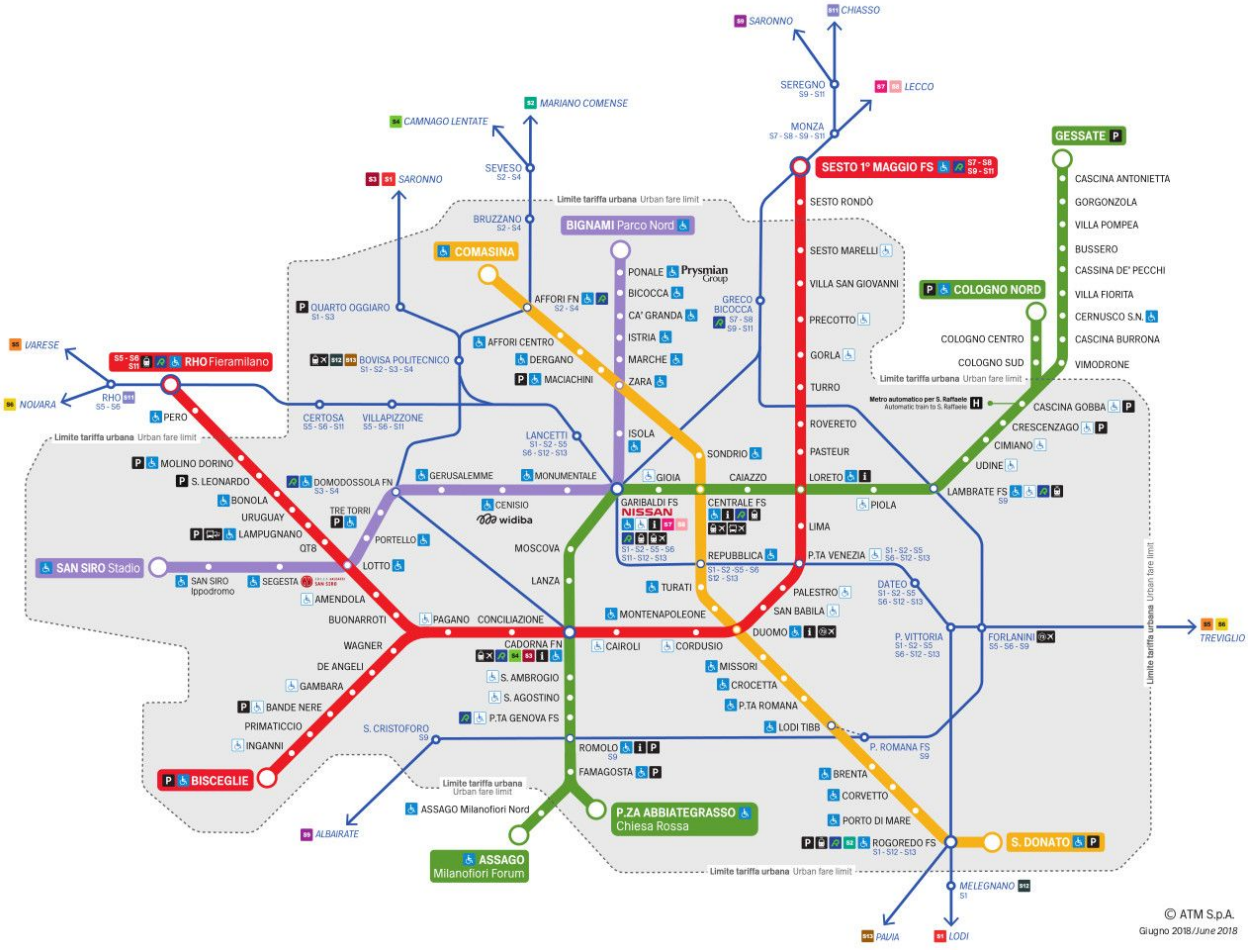

Figure 6.1 - Map of the Milan Subway Line (ATM)

Distance traveled with hybrid subway deliveries by car operators (km)	Distance traveled by trucks with traditional ground logistics (km)	Reduction in distance traveled with ground vehicles
5.2	25.1	4.83
7.8	25.7	3.29
5.5	19.3	3.51
6.3	24.9	3.95
5.2	22.8	4.38
6.2	18.4	2.97
		3.82

Table 6.1 - Reduction in road distance traveled using hybrid subway deliveries

Time with hybrid subway deliveries (min)	Time with traditional ground deliveries (min)	Improvement time (%)
59.67	84.13	29.07%
41.47	58.12	28.65%
32.3	60.93	46.99%
19.28	61.88	68.84%
44.18	75.25	41.29%
21.53	28.97	25.68%
42.45	81.18	47.71%
18.52	36.87	49.77%
46.02	73.23	37.18%
42.35	81.38	47.96%

Table 6.2 - Time improvement for simulated random deliveries in Milan

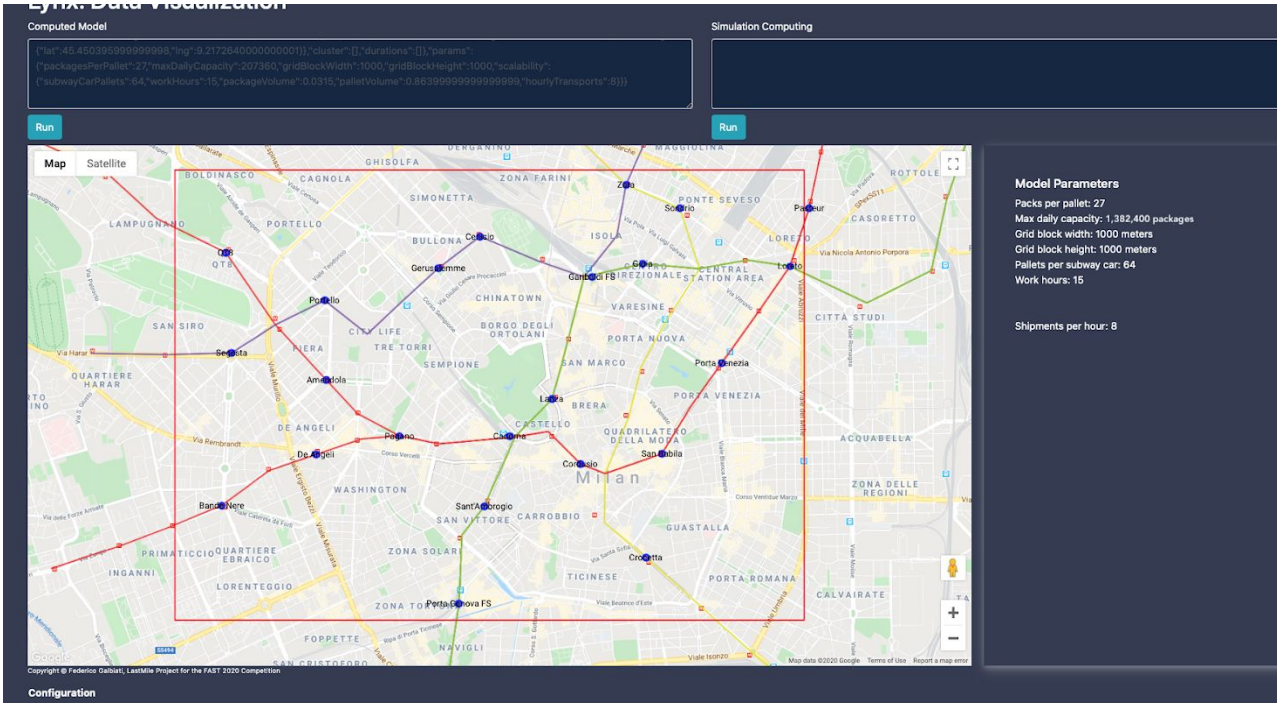

Figure 6.2 - GUI Showing the selected logistics subway terminals

7. Discussion

This delivery logistics system based on subways was demonstrated optimal to perform express deliveries in cities. The model allows a reduction in the number of trucks on the streets, improved safety on roads, and lower environmental impact. Moreover, this system could be fundamental for time-sensitive services. For example, pharmacies could order medications from warehouses and have them delivered within thirty minutes to an hour, removing the traditional one day ordering times. Morning deliveries of newspapers to newsstands and shops could also rely on this hybrid logistics model. The capillarity of the subway infrastructure could allow reaching the entirety of a city rapidly and sustainably. Moreover, libraries could share books with each other to provide a larger selection of content. Today, an increasingly popular business model is the removal of inventories from stores and the delivery of purchased items to the customer directly. Such retail concept was successfully implemented by Nordstrom [14, 15], whose customers usually receive the orders the day after ordering. This subway-based logistic system would enable removing shop inventories while providing fast same-day deliveries. Nowadays, services that require fast deliveries are becoming increasingly popular and this logistic model can be the solution. Moreover, compared to technologies such as drones, which are still in a newborn stage, hybrid subway-based deliveries can be a reality with current technologies and provide reliable service to millions of worldwide citizens each day.

8. Conclusions

This research proved feasible to utilize already established urban subway systems to solve the last-mile delivery issue through express deliveries (less than one hour). The system, moreover, would be capable of managing a city such as Milan even during the winter holiday season, when delivery demand sharply increases. The modeling and real-time simulation using powerful algorithms optimized this solution. It minimized the number of terminal stations and delivery times, also based on traffic info, location in a grid, and prioritization of stations on the same line. Moreover, due to the small number of packages to transport in each batch, minicars or electric vehicles could be used, to further prevent the environmental, safety, and traffic issues posed to a city. Such a method could be useful not just for e-commerce deliveries, but potentially express services, including commercial, healthcare, and largely distributed activities. The hybrid subway

delivery system was found able to transfer up to about 1.4 million daily packages (mix of cardboard boxes and envelopes) per line. Moreover, it could fulfill the one-hour delivery requirement and complete some deliveries in less than 30 minutes. Additionally, the system showed a reduction by four times of CO_2 emission and an increase in the operator time availability between 30% and 70% in comparison to traditional ground logistics.

9. Acknowledgments

I would like to thank Domenico Latanza, Computer Science Teacher at Liceo Scientifico Donato Bramante, for motivating me to pursue more Software Engineering projects.

10. References

[1] "Breadth First Search, Dijkstra's Algorithm for Shortest Paths." *Grainger College of Engineering*, courses.engr.illinois.edu/cs473/sp2011/Lectures/03_class.pdf.

[2] "Combining the Intelligent Design of the Subway in the Field of Express Logistics." *IDEEA*, ideea19.org/index.html/attachments/article/15/Combining the intelligent design of the subway in the field of express logistics.pdf.

[3] "Connecting with Shoppers in the Age of Choice." *Narvar*, see.narvar.com/rs/249-TEC-877/images/Connecting With Shoppers_Narvar Consumer Report Q1 2018.pdf?aliId=1109557.

[4] "E-Commerce Statistics for Individuals." *European Commission*, ec.europa.eu/eurostat/statistics-explained/pdfscache/46776.pdf.

[5] Gatta, Valerio, et al. "Sustainable Urban Freight Transport Adopting Public Transport-Based Crowdshipping for B2C Deliveries." *European Transport Research Review*, vol. 11, no. 1, Springer Science and Business Media LLC, Feb. 2019. Crossref, doi:10.1186/s12544-019-0352-x.

[6] "Get Traffic Flow Segment (Azure Maps)." *Microsoft Azure Docs*, docs.microsoft.com/en-us/rest/api/maps/traffic/gettrafficflowsegment.

[7] "Green Logistics in Last Mile Delivery (B2C E-Commerce)." *Polimi*, www.politesi.polimi.it/bitstream/10589/137999/3/Thesis - Mohamed Saleh.pdf.

[8] "How the Genetic Algorithm Works." *MATLAB*, www.mathworks.com/help/gads/how-the-genetic-algorithm-works.html.

[9] Kikuta, Jun, et al. "New Subway-Integrated City Logistics Szystem." *Procedia - Social and Behavioral Sciences*, vol. 39, Elsevier BV, 2012, pp. 476–489. Crossref, doi:10.1016/j.sbspro.2012.03.123.

[10] "Last Mile Logistics - TRIMIS - European Commission." *TRIMIS*, 10 Oct. 2019, trimis.ec.europa.eu/project/last-mile-logistics.

[11] "Milano Leonardo Linee 1-2." *Hitachi Rail*, italy.hitachirail.com/milano-leonardo-linee-1-2_393.html.

[12] "Retail Travelution." *Addleshaw Goddard*, www.addleshawgoddard.com/globalassets/insights/transport/retail-travelution-retail-logistics.pdf.

[13] Serafini, Simone, et al. "Sustainable Crowdshipping Using Public Transport: A Case Study Evaluation in Rome." *Transportation Research Procedia*, vol. 30, Elsevier BV, 2018, pp. 101–110. Crossref, doi:10.1016/j.trpro.2018.09.012.

[14] "The Future of Retail 2018." *Walker Sands*, www.walkersands.com/wp-content/uploads/2018/07/Walker-Sands_2018-Future-of-Retail-Report.pdf.

[15] Thomas, Lauren. "Nordstrom Plans to Open More Stores with No Inventory in Los Angeles, New York." *CNBC*, CNBC, 9 July 2018,

www.cnbc.com/2018/07/09/nordstrom-to-open-more-no-inventory-stores-in-los-angeles-new-york.html.
[16] “Traveling Salesman Problem | OR-Tools | Google Developers.” Google, developers.google.com/optimization/routing/tsp.
[17] “Why the Last Mile of a Delivery Matters Most.” Honeywell, www.honeywell.com/en-us/newsroom/news/2019/10/why-the-last-mile-of-a-delivery-matters-most.

8. Appendix

Below are some code snippets that were created, including the algorithms and Lynx web platform API.

```
// ================================  MODEL COMPUTING  ================================

let GoogleMaps = GoogleMapsAPI()
// Import the list of subway stations with locations
let osjson = OverpassSubwayJSON()
let opStations = osjson.parse(city: .Milan)
// Import YAML subway sequence details
let OSMSParser = OSMSAPIParser()
let osmSubwayRoot = OSMSParser.parse(city: .Milan)

let areaInterest = GeoArea(name: "Milan, Italy",
                           area: GeoSection(topLeft: GeoPoint(latitude: 45.494,
                                                              longitude: 9.13),
                                            bottomRight: GeoPoint(latitude: 45.450396,
                                                                  longitude: 9.217264)),
                                            gridBlockWidth: 1000,
                                            gridBlockHeight: 1000)

var lynxModel = WALogisticModel(stations: [],
                                polylines: [],
                                square: WASquare(geoSection: areaInterest.geoSection),
                                cluster: [],
                                durations: [],
                                carDurations: [])

let paramFramework = LogisticsScalabilityFramework(area: areaInterest,
                                                   params: ScalabilityParams(subwayCarPallets: 64,
                                                                             hourlyTransports: 8))

// Create a grid with stations
let grid = areaInterest.gridSubway(opStations)

// Set WA model from OSMS and OP
lynxModel.set(grid: grid)
lynxModel.set(stations: opStations, routes: osmSubwayRoot.routes)

// Create a JSON file for the WebApp
let waencoder = WebAppJSONEncoder(model: lynxModel, params: paramFramework.computeTotalDailyCapacity())
let waData = waencoder.encode()
```

Modeling the logistics network

```
// ================================  SIMULATION COMPUTING  ================================

var simulationModel = WALogisticModel(stations: [],
                                      polylines: [],
                                      square: WASquare(geoSection: areaInterest.geoSection),
                                      cluster: [],
                                      durations: [],
                                      carDurations: [])
var db = Database()
var unitTests = UnitTestsDeliveries()
var deliveries: [Delivery] = unitTests.tests()

for delivery in deliveries {
    db.addOrder(order: delivery)
}

let destinationsEdges = db.computeTargetStations()
for edges in destinationsEdges {
    simulationModel.set(edges: edges)
}

var durations: [Int] = []
var carDurations: [Int] = []
for (deptStation, cluster) in db.targetStations {
    guard let fromStation = opStations.find(db.warehouse) else { break }
    let point = GeoPoint(latitude: deptStation.geometry.lat, longitude: deptStation.geometry.lon)
    let from = GeoPoint(latitude: fromStation.geometry.lat, longitude: fromStation.geometry.lon)
    GoogleMaps.requestDirections(origin: from.toString(),
                                 destination: point.toString(),
                                 mode: .transit,
                                 transitMode: .subway) { (directions) in
        if let route = directions.routes.first {
            durations.append(route.totalDuration)
        }
    }
    GoogleMaps.requestDirections(origin: from.toString(),
                                 destination: point.toString(),
                                 mode: .driving,
                                 transitMode: .none) { (directions) in
        if let route = directions.routes.first {
            carDurations.append(route.totalDuration)
        }
    }
    let destinations = cluster.map({ (d) -> GeoPoint in
        return d.address.geoPoint
    })
    simulationModel.set(coordinates: [point] + destinations + [point])
}
simulationModel.set(durations: durations)
simulationModel.set(carDurations: carDurations)

simulationModel.set(nodes: db.destinationStations)

// Create a JSON file for the WebApp
let waencoder2 = WebAppJSONEncoder(model: simulationModel, params: paramFramework.computeTotalDailyCapacity())
let simData = waencoder2.encode()
```

Computing the delivery terminals for a simulation

```
public func gridSubway(_ overpassData: OverpassData) -> [[GridSlot]] {
    var stations: Set<OverpassFeature> = Set(overpassData.features)
    let sectionsHorizontal = Int((geoSection.width / gridBlockWidth).rounded(.up))
    let sectionsVertical = Int((geoSection.height / gridBlockHeight).rounded(.up))

    let lines = Array(repeating: GridSlot(), count: sectionsHorizontal)
    let columns = Array(repeating: lines, count: sectionsVertical)
    var grid: [[GridSlot]] = columns

    var yPosOffset = 0.0
    for row in 0..<columns.count {
        var xPosOffset = 0.0
        for column in 0..<lines.count {
            let newSection = GeoSection(topLeft: geoSection.topLeft.offsetBy(dn: -yPosOffset, de: xPosOffset),
                                        bottomRight: geoSection.topLeft.offsetBy(dn: -(yPosOffset + gridBlockHeight),
                                                                                 de: xPosOffset + gridBlockWidth))
            let newGridBlock = GridSlot(station: nil, geoData: newSection)
            let results = stationsInSection(stations: stations, geoSection: newSection)
            if results.count > 0 {
                if let areaStation = bestStationForArea(stations: results, geoSection: newSection) {
                    stations.remove(areaStation)
                    newGridBlock.station = areaStation
                }
            }
            grid[row][column] = newGridBlock
            xPosOffset += gridBlockWidth
        }
        yPosOffset += gridBlockHeight
    }
    self.grid = grid
    return grid
}
```

Creating a grid and setting their parameters

```swift
public class GoogleMapsAPI {
    private func reformatURLSpaces(_ param: String) -> String {
        return param.replacingOccurrences(of: " ", with: "%20")
    }
    public func requestDirections(origin: String, destination: String,
                                  mode: TravelMode, transitMode: TransitMode = .none,
                                  completion: @escaping (_ value: GMapsDirections) -> Void) {
        var urlString = GoogleMapsAPIParams.directionsEndPoint
        urlString += "origin=\(reformatURLSpaces(origin))&"
        urlString += "destination=\(reformatURLSpaces(destination))&"
        urlString += "mode=\(mode.rawValue)&"
        urlString += "transit_mode=\(transitMode.rawValue)&"
        urlString += "key=\(GoogleMapsAPIParams.API_KEY)&"
        urlString += "departure_time=\(simulationDepartingTime)"
        guard let url = URL(string: urlString) else { return }

        makeRequest(to: url, gMapsDecoderType: .directions) { (structure) in
            if let directions = structure as? GMapsDirections {
                completion(directions)
            }
        }
    }
    public func geocoding(address: Address, completion: @escaping (_ value: GMapsGeocoding) -> Void) {
        var urlString = GoogleMapsAPIParams.geocodingEndPoint
        urlString += "key=\(GoogleMapsAPIParams.API_KEY)&"
        urlString += "address=\(reformatURLSpaces(address.toString()))"
        guard let url = URL(string: urlString) else { return }
        makeRequest(to: url, gMapsDecoderType: .geocoding) { (structure) in
            if let stations = structure as? GMapsGeocoding {
                completion(stations)
            }
        }
    }
    private func makeRequest(to url: URL, gMapsDecoderType: GMapsDecoderType,
                             completion: @escaping (_ value: Decodable) -> Void) {
        var done = false, dataReturn: Data?
        let task = URLSession.shared.dataTask(with: url) { (data, response, error) in
            if let error = error { print(error) }
            guard let data = data else { print("Error response or data missing"); return }
            dataReturn = data
            done = true
        }
        task.resume()

        repeat {
            RunLoop.current.run(until: Date(timeIntervalSinceNow: 0.1))
        } while !done

        guard let dataResponse = dataReturn else { print("Error, dataresponse invalid"); return }
        do {
            let decoder = JSONDecoder()
            switch gMapsDecoderType {
            case .places:
                let response = try decoder.decode(GMapsPlaceQuery.self, from: dataResponse)
                completion(response)
            case .directions:
                let response = try decoder.decode(GMapsDirections.self, from: dataResponse)
                completion(response)
            case .geocoding:
                let response = try decoder.decode(GMapsGeocoding.self, from: dataResponse)
                completion(response)
            }
        } catch {
            print(error)
        }
    }
}
```

Google Maps code snippet for making requests

```
public struct WALogisticModel: Encodable {
    var stations: [WAStation]
    var polylines: [WAPolyline]
    var square: WASquare
    var cluster: [[WACoordinates]]
    var durations: [Int]
    var carDurations: [Int]
    mutating func set(grid: [[GridSlot]]) {
        for row in grid {
            for col in row {
                if let station = col.station {
                    stations.append(WAStation(overpassFeature: station))
                }
            }
        }
    }

    mutating func set(stations: OverpassData, routes: [OSMSLine]) {
        for route in routes {
            for itinerary in route.itineraries {
                var polylinesLatLng: [WACoordinates] = []
                for OSMSstation in itinerary.value {
                    var found = false
                    for overpassStation in stations.features {
                        if OSMSstation.contains(overpassStation.properties.name ?? "") {
                            let lineCoordinates = WACoordinates(lat: overpassStation.geometry.lat,
                                                                lng: overpassStation.geometry.lon)
                            polylinesLatLng.append(lineCoordinates)
                            found = true
                            break
                        }
                    }
                    if !found {
                        print ("ERROR = Could not find \(OSMSstation)")
                    }
                }
                polylines.append(WAPolyline(coordinates: polylinesLatLng, color: route.colour, thickness: 2))
            }
        }
    }

    mutating func set(edges: [Edge<String>]) {
        for edge in edges {
            let n1 = edge.source.data.split(separator: "(").first!.trimmingCharacters(in: .whitespacesAndNewlines)
            let n2 = edge.destination.data.split(separator: "(").first!.trimmingCharacters(in: .whitespacesAndNewlines)
            guard let s1 = opStations.find(n1), let s2 = opStations.find(n2) else { break }
            setPolyline(s1: s1, s2: s2)
        }
    }

    mutating func set(nodes: Set<OverpassFeature>) {
        for s in nodes {
            stations.append(WAStation(overpassFeature: s, color: "eb5b7d"))
        }
    }

    mutating func setPolyline(s1: OverpassFeature, s2: OverpassFeature) {
        let lineCoordinates = [WACoordinates(lat: s1.geometry.lat,
                                             lng: s1.geometry.lon),
                               WACoordinates(lat: s2.geometry.lat, lng: s2.geometry.lon)]

        let poly = WAPolyline(coordinates: lineCoordinates, color: "4f00fc", thickness: 10)
        polylines.append(poly)
    }

    mutating func set(durations: [Int]) {
        durations.forEach { (time) in
            self.durations.append(time)
        }
    }
```

Lynx JSON model and data creation

```
public class OSMSAPIParser {
    func parse(city: CityDatasets) -> OSMSRoot {
        let url = URL(fileURLWithPath: sysPath + "/OSMSYAML/\(city.rawValue).yaml")
        let data = try! Data(contentsOf: url)
        let yamlString = String(data: data, encoding: .utf8)!
        let decoder = YAMLDecoder()
        let decoded = try! decoder.decode(OSMSRoot.self, from: yamlString)
        return decoded
    }
}


public struct OSMSRoot: Codable {
    let stations: [String]
    let routes: [OSMSLine]
    let transfers: [[String]]

    func find(_ station: String) -> String {
        for osmStation in stations {
            if osmStation.contains(station) {
                return osmStation
            }
        }
        return ""
    }
}

public struct OSMSLine: Codable {
    let ref: String
    let name: String
    let colour: String
    let stations: [String]
    let itineraries: [String : [String]]
    let station_count: Int
}
```

Open Street Map Subway YAML Parsing

```
public struct OverpassData: Codable {
    var features: [OverpassFeature]

    func find(_ station: String) -> OverpassFeature? {
        for OPStation in features {
            if let name = OPStation.properties.name {
                if name.contains(station) {
                    return OPStation
                }
            }
        }
        return nil
    }
}

public struct OverpassFeature: Codable, Hashable {
    let properties: OverpassProperty
    let geometry: OverpassGeometry
    public static func == (lhs: OverpassFeature, rhs: OverpassFeature) -> Bool {
        if lhs.properties.id == rhs.properties.id {
            return true
        }
        return false
    }

    public func hash(into hasher: inout Hasher) {
        hasher.combine(properties.id)
    }
}

public struct OverpassGeometry: Codable {
    let type: String
    let coordinates: CoordinatesValue
    var lat: Double { return coordinates.coordinates[1] }
    var lon: Double { return coordinates.coordinates[0] }
}

struct CoordinatesValue: Codable {
    let coordinates: [Double]
    let coordinatesSets: [[Double]]

    init(from decoder: Decoder) throws {
        let container =  try decoder.singleValueContainer()
        do {
            coordinates = try container.decode([Double].self)
            coordinatesSets = []
        } catch {
            coordinatesSets = try container.decode([[Double]].self)
            coordinates = []
        }
    }

    func encode(to encoder: Encoder) throws {
        var container = encoder.singleValueContainer()
        if coordinates.count != 0 {
            try container.encode(coordinates)
        } else if coordinatesSets.count != 0 {
            try container.encode(coordinatesSets)
        }
    }
}

public struct OverpassProperty: Codable, Hashable {
    let id: String?
    let name: String?
    let nameen: String?
    let ref: String?
    let stroke: String?

    private enum CodingKeys: String, CodingKey {
        case id = "@id"
        case name
        case nameen = "name:en"
        case ref
        case stroke
    }
}
```

Overpass JSON Parsing