Stable Matching for Dynamic Ridesharing

Prescriptive Analysis Final Report

IEDA3010

Victoire Anguelidis (20895724), Kamran Arshad Butt (20891443), Nikhil Joseph (20895645), Sai Vineeth (20916994), Raghav Shukla (20887612)

Table of Contents

1. Introduction

1.1. Background/Case Details

Over recent years, advancements in technology and a multitude of innovations have led to a significant increase in the utilization of ridesharing services. This surge in popularity, coupled with the substantial profitability of the sector, underscores the necessity for methods that enhance efficiency within on-demand ridesharing operations.

Ridesharing optimization offers numerous benefits by improving the efficiency and sustainability of transportation systems. By matching riders and drivers more effectively, it reduces travel times, minimizes fuel consumption, and lowers operational costs for service providers. Additionally, it helps decrease traffic congestion and carbon emissions, contributing to a greener and more sustainable urban environment. Optimized ridesharing also enhances user satisfaction by reducing wait times and improving reliability, making it a more attractive alternative to private vehicle ownership. Ultimately, ridesharing optimization supports smarter, more eco-friendly mobility solutions in growing cities.

This report aims to develop a ridesharing model that efficiently matches riders with drivers. Our objective is to evaluate and compare different weighting methods for assessing matches and to identify the algorithms that deliver the highest performance. Additionally, we present the binomial linear programming derivations that form the foundation of our model. For this analysis, we utilize Melbourne traffic data sourced from the Australian Bureau of Statistics and follow the methodology outlined in the paper *"Novel Dynamic Formulations for Real-Time Ride-Sharing Systems"* by Ali Najmi, David Rey, and Taha H. Rashidi (2017).

1.2. Objective

Our primary objective is to implement the binary integer linear optimization algorithm described in the paper referenced in Section 1.1. Specifically, we aim to demonstrate how the mathematical formulations presented in the paper can be translated into executable code, enabling us to conduct large-scale simulations on the dataset provided. This process involves carefully converting theoretical models into a computational framework that can handle real-world data, ensuring that the algorithm functions effectively in practical scenarios.

Our secondary objective is to evaluate and compare the different weighting methods discussed in the paper to determine their performance across various metrics. By conducting a multivariate analysis, we assess the performance of these methods across different sample sizes and user densities. This provides valuable insights into how ridesharing algorithms perform under diverse conditions, offering practical guidance for optimizing ridesharing systems. This objective is particularly significant, as it highlights the trade-offs and efficiencies of different weighting approaches, ultimately contributing to the development of more robust and scalable ridesharing solutions.

2. Model Building

2.1. Assumptions and Variables

The dataset provided for this study is structured as follows: each row represents a separate announcement, either from a driver or a rider. Announcements are organized in ascending order based on the *Announcement* column, which corresponds to the Trip Request ID. Driver announcements are assigned IDs less than 100,000, while rider announcements are assigned IDs greater than or equal to 100,000.

Column Name: Description

- *Announcement:* Trip Request ID.
- *Origin:* The SLA (Statistical Local Area) code of the origin location.
- *Destination:* The SLA code of the destination location.
- *Distance_Car-Peak:* The distance (in kilometers) between the origin and destination by car during peak hours.
- *Time_Car-Peak:* The travel time (in minutes) between the origin and destination by car during peak hours.
- *Announcementtime:* The time when the announcement is entered into the system.
- *Starttime:* The preferred departure time (in minutes), where 0 corresponds to 12:00 AM.
- *Earliesttime:* The earliest allowable departure time (in minutes), where 0 corresponds to 12:00 AM.
- *Latesttime:* The latest allowable arrival time (in minutes), where 0 corresponds to 12:00 AM.
- *Origin_Latitude:* The latitude coordinates of the origin point.
- *Origin_Longitude:* The longitude coordinates of the origin point.
- *Destination_Latitude:* The latitude coordinates of the destination point.
- *Destination_Longitude:* The longitude coordinates of the destination point.

It is important to note that the datasets provided for this study, consisting of CSV files spanning hundreds of thousands of rows, exceeded the processing capabilities of our hardware. As a result, we were limited to analyzing datasets containing only a few hundred rows of announcements at a time. To ensure consistency and manageability, we selected an equal number of driver and rider announcements from the dataset, regardless of the original ratio of driver to rider announcements. For instance, we sampled 50, 100, 200, and so on, of driver and rider announcements each. In a real-world context, this approach assumes an equal number of drivers and riders seeking matches within a given area, such as a city.

In the later sections of this report, geographic distances between points are required for analysis. The dataset provided includes a column containing SLA codes for each announcement. SLA codes are part of a proprietary zoning system used by the city of Melbourne. However, due to the lack of publicly available resources explaining how to interpret these codes, we opted to utilize the longitude and latitude information provided for each announcement. These coordinates were used to calculate the Euclidean distance between any two points, enabling us to approximate geographic distances effectively.

It is important to note that this method carries an implicit assumption, as it ignores the complexities of urban travel. In reality, travel within cities cannot occur in a straight line due to road networks, traffic patterns, and other geographical constraints. For our analysis, we used the geopy library to calculate distances based on geographic coordinates. While this approach provides a general approximation, there are alternative libraries and tools available that incorporate detailed urban data, such as road networks and traffic conditions, to calculate more accurate distances between locations. Incorporating these tools in future iterations could significantly enhance the accuracy and realism of the model.

2.2. Evaluation Metrics

The performance of our model is evaluated using two key metrics. The first metric is the Matching Rate (MR), calculated as the ratio of successfully matched pairs to the total number of driver and rider announcements. This metric solely measures the quantity of matches achieved and does not account for additional factors such as distance, suitability, or other quantitative/qualitative characteristics of the matches. This is calculated as follows:

The matching rate (*MR*) can be expressed using the following equation:

$$MR = \frac{2 \times (\text{Total Number of Matched Pairs})}{\text{Number of Driver Announcements} + \text{Number of Rider Announcements}}$$

The second metric used to evaluate the performance of the model is Average Kilometers Saved (AKS). This metric quantifies the reduction in total driving distance achieved by matching a driver and rider, compared to the total distance if they had each taken their trips independently. It is calculated by summing the distances of the driver's and rider's individual trips and subtracting the total distance of the shared trip taken by the driver.

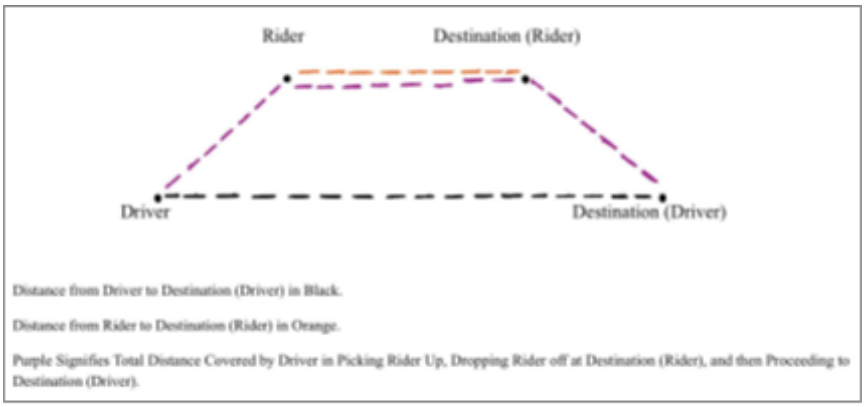


Figure 2.2.A: Average Kilometers Savings (AKS) calculations

Kilometers Saved = (Driver's Individual Trip Distance + Rider's Individual Trip Distance) − Shared Trip Distance

This calculation is applied to every single match the model makes and then the average kilometer saved across all matches is calculated from this set of values. Note that it is possible to have a negative AKS. This is for cases where the driver has to go so far out of his way to pick up and drop the rider that the shared trip is longer than if both had gone their separate ways.

2.3. Pre-Processing & Feasible Pairs

The first step in processing the CSV data involves separating the rider and driver announcements into two distinct dataframes. Subsequently, we extract the first 50, 100, 200, or more rows, depending on the sample size selected for the specific iteration of the model. At this stage, the driver and rider DataFrames each contain 13 columns for each of the variables outlined in Section 2.1, along with the number of rows corresponding to the chosen sample size. Note that for this example, our driver matrix has dimensions of 100x13.

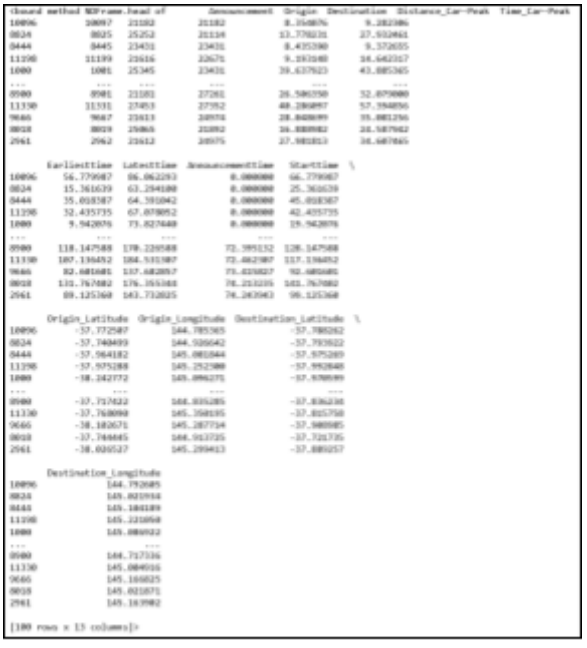


Figure 2.3.A: Initial extracted list of driver announcements

The next step involves performing a cross-join between the driver and rider dataframes to generate a new table containing all possible pairings of driver and rider announcements. This results in a Cartesian product of the two datasets. For instance, if we begin with 100 driver announcements and 100 rider announcements, the resulting table will contain 10,000 rows, representing every possible combination of driver-rider pairs. Each pair is uniquely identified by the 'Combined_Announcement' column which is a combination of the driver and rider ids in that order:

Note that after the cross join, starting with 100 driver and 100 rider announcements, we are left with 10,000 pairings, with 27 columns: 13 from the driver, 13 from the rider, and 'Combined_Announcement' as key value for each row.

Announcement_x: 10097 + Announcement_y: 101070 = Combined_Announcement: 10097101070

The next step in the process is to eliminate infeasible driver-rider pairs. For instance, if a driver's earliest departure time from their origin occurs after the rider's latest allowable arrival time at their destination, it is not feasible to match these announcements. Such pairs are excluded from further consideration and are not included in the optimization problem. The mathematical condition used to determine feasibility is as follows:

$$k = \min \begin{Bmatrix} \text{Latest Rider Arrival Time} - \text{Travel Time}(\text{Origin (r)} \to \text{Destination (r)}) \\ - \text{Travel Time}(\text{Origin (d)} \to \text{Origin (r)}) \\ \text{Latest Driver Arrival Time} - \text{Travel Time}(\text{Destination (d)} \to \text{Destination (r)}) \\ - \text{Travel Time}(\text{Origin (d)} \to \text{Origin (r)}) \end{Bmatrix}$$

1. **For the Driver:**
   The match is feasible if the difference in the driver's travel time is greater than or equal to 0:

$$k - \max(\text{Current Time}, \text{Earliest Driver Departure Time}) \geq 0$$

2. **For the Rider:**
   The match is feasible if the difference in the rider's travel time is greater than or equal to 0:

$$k + \text{Travel Time}(\text{Origin (d)} \to \text{Origin (r)}) - \max(\text{Current Time}, \text{Earliest Rider Departure Time}) \geq 0$$

Figure 2.3.C: The feasibility conditions to determine whether the rider and driver announcements should be paired

After applying this condition, the number of feasible pairs to consider—and consequently, the number of decision variables to optimize—has been reduced to 5,079 down from 10,000 in the sample being analyzed in this report.

3. Static Model

3.1. Binary Programming

To construct the static model, the method employed in the referenced paper formulates the problem as a binary optimization problem. For each of the feasible paths identified in the earlier steps, a binary decision variable is assigned: a value of '1' indicates that the path is selected as a final matched pair, while a value of '0' indicates that it is not selected. Consequently, given that the data frame of feasible pairs contains 5,079 rows, the model will include 5,079 binary decision variables to optimize. The full optimization problem can be outlined as follows:

- Let x_{dr} be a binary decision variable, where $x_{dr} = 1$ if the driver d and rider r are matched, and $x_{dr} = 0$ otherwise.
- Let $w_{dr} \geq 0$ denote the weight representing the contribution of matching driver d with rider r in the objective function.

Objective Function:

$$\textbf{Maximize} \sum_{(d,r) \in P'} w_{dr} \cdot x_{dr}$$

Subject To:

$$\sum_{r \in R:(d,r) \in P'} x_{dr} \leq 1 \quad \forall d \in D$$

$$\sum_{d \in D:(d,r) \in P'} x_{dr} \leq 1 \quad \forall r \in R$$

The maximization function computes the sum of the product of the binary decision variable for each pair and the corresponding weight assigned to that pair. The weight serves as a measure of how advantageous or suitable a particular pairing is, with different weighting methods being discussed in subsequent sections. If a pairing is selected (i.e., the binary variable equals 1), the contribution of that pair to the summation is equal to the value of the weight assigned to it. Conversely, if the pairing is not selected (i.e., the binary variable equals 0), the contribution of that term to the summation is 0. The summation sums across all possible pairings of driver and rider in the dataframe of feasible matches.

The two constraints simply ensure that each driver is paired with only one rider and each rider with only one driver.

3.2. Weighting Strategies

There are a number of different weighting strategies that can be used to compare pairings. We have examined four:

1. *Maximizing total number of matches*
   This is critical for the ride-sharing service to be sustainable. It can be used as a measure of reliability. This means that every match is valued equally, regardless of how convenient/suitable the match is. It would be with uniform weights $w_{dr} = 1$

$$\textbf{Maximize} \sum_{(d,r) \in P'} x_{dr}$$

2. *Maximizing net distance savings*
   This approach utilizes calculations similar to those employed in the *Average Kilometers Saved* evaluation metric. Specifically, it determines the distance saved by matching a driver and rider for a shared trip, as opposed to each completing their trips independently.

   In cases where the two trips are geographically distant from one another, a negative weight may be applied. This occurs when the total distance required for the driver to reach the rider's starting point, complete their trip, and then proceed to the driver's destination exceeds the distance the driver would have traveled independently. Consequently, this method is more selective in identifying feasible and efficient matches compared to the first evaluation metric.

   $\Delta S(d,r) =$ *(Driver's Individual Trip Distance + Rider's Individual Trip Distance) − Shared Trip Distance*

$$\textbf{Maximize} \sum_{(d,r) \in P'} \Delta S(d,r) \cdot x_{dr}$$

3. *Maximizing total distance proximity index*

   This approach assigns weights based on how similar the lengths are of the driver's and rider's trips. The Distance Proximity (DP) is calculated below:

$$DP(d,r)=\min\left(\frac{Driver's\ trip\ length}{Rider's\ trip\ length},\frac{Rider's\ trip\ length}{Driver's\ trip\ length}\right)$$

$$\textbf{Maximize} \sum_{(d,r)\ \epsilon\ P'} DP(d,r)\cdot x_{dr}$$

   Since the smaller ratio is always selected, the resulting value of **DP(d, r)** is always between 0 and 1. The value of **DP(d, r)** approaches 0 as the difference in trip lengths increases, and it approaches 1 as the trip lengths become more similar.

4. *Maximizing total adjusted distance proximity index*

   This approach builds off of the previous one, but adds another term in the weightage calculation that adjusts for the extra distance travelled by the driver to facilitate the rideshare. A notable limitation of the previous unadjusted total distance proximity weighting method is that, while it optimizes for trip length similarity, it does not account for the start and end locations of the trips. This means that two trips with significantly different origins and destinations may still receive a high weight if their lengths are similar, potentially leading to suboptimal pairings in terms of geographic feasibility. This is adjusted in this weighing method, which adds an Adjustment Factor (AF) calculated as below:

$$AF(d,r)=\frac{Driver's\ trip\ length}{Total\ shared\ trip\ length}$$

   This adjustment factor rewards matches where the additional distance traveled due to the rider is small relative to the driver's original trip. It then adds this term to the overall summation:

$$\textbf{Maximize} \sum_{(d,r)\ \epsilon\ P'} AF(d,r)\cdot DP(d,r)\cdot x_{dr}$$

3.3. Python Implementation

The implementation of the model in Python differs slightly based on the weighting method utilized. For this purpose, the weighting methods have been classified into two categories: *equal weightage* and *unequal weightage*.

3.3.1 Equal Weightage

Only one weighting method employs equal weightage, which is the total number of matches maximization method. In this approach, every feasible match is assigned a weight of 1. As a result, the objective function, when implemented in Python using the gurobi.py library, consists of a single term within the summation: $\mathbf{x_{dr}}$.

```
mode=Model('maximizer')
x=model.addVars(possible_matches,vtype=GRB.BINARY, name="x") Declaring decision variables (x_dr for all d,r)
model.setObjective(x.sum(),GRB.MAXIMIZE) Declaring Objective Function


for d in drivers:
    model.addConstr(sum(x[d,r] for r in riders if (d,r) in x)<=1)

for r in riders:                                                  } Declaring one-to-one driver to rider match constraints
    model.addConstr(sum(x[d,r] for d in drivers if (d,r) in x)<=1)
model.optimize()
```

Figure 3.3.1.A: Code to define objective function and constraints for equal weightage problem

3.3.2 Unequal Weightage

For each of the subsequent three weighting methods, a 28th column was added to the feasible set dataframe generated at the conclusion of Section 2.3. This column stores the weight calculated for each pairing represented by

the corresponding row. Depending on the weighting method selected, the values in this column were initially set to zero and then updated in accordance with the specific calculations required by the chosen method.

Each time the model is run, after the steps completed in section 2.3, a new column is declared for the dataframe, and every value inside the column is set to zero. Then, via a loop, the weight of each row (pairing) is calculated and replaces the zero in that row. Now there is a weight vector that the optimization model can use as input to generate the optimal pairing solution.

```
mode=Model('maximizer')
x=model.addVars([(d,r,weight) for d,r,weight in possible_matches],vtype=GRB.BINARY, name="x")
model.setObjective(sum(x[d,r,weight]*weight for d,r,weight in possible_matches),GRB.MAXIMIZE)


for d in drivers:
    model.addConstr(sum(x[d,r,weight] for driver_match,r,weight in possible_matches if driver_match==d)<=1)

for r in riders:
    model.addConstr(sum(x[d,r,weight] for d,rider_match,weight in possible_matches if rider_match==r)<=1)
model.optimize()
```

Figure 3.3.2.B: Code to define objective function and constraints for unequal weightage problem

The definition of the model using the gurobi library also varies slightly when adding weights to each term. While the objective function differs by including a weight term within the summation, the constraints remain unchanged. It is important to note that, although the code syntax may differ slightly, the underlying functionality remains consistent.

4. Static Problem Solutions

4.1 Static Model Solutions and Interpretations

Due to hardware constraints, it was necessary to use significantly smaller sample sizes. To address this, a multivariate analysis was conducted, comparing the model's performance across different sample sizes and weighting methods. The evaluation metrics described in Section 2.2 were applied after the binary optimization problem had been solved:

```
#performance evaluation terminal

#matchrate calc
matched2=0
for dr2 in possible_matches:
    if x[dr2].x>0.5:
        matched2=matched2+1
matchrate=matched2*2/(len(drivers)+len(riders))
print(matchrate)

#Average Kilometer Saved (AKS) calc
kstotal=0
matched3=0
for dr in possible_matches:
    if x[dr].x>0.5:
        drivertrip = driverdf[driverdf['Announcement'] == dr[0]]['Distance_Car-Peak']
        drivertriplength=float(drivertrip)
        ridertrip = riderdf[riderdf['Announcement'] == dr[1]]['Distance_Car-Peak']
        ridertriplength=float(ridertrip)
        nomatchlength=drivertriplength+ridertriplength

        driverrow=driverdf[driverdf['Announcement'] == dr[0]]
        driverorigin=(float(driverrow['Origin_Latitude']),float(driverrow['Origin_Longitude']))
        driverendpoint=(float(driverrow['Destination_Latitude']),float(driverrow['Destination_Longitude']))

        riderrow=riderdf[riderdf['Announcement'] == dr[1]]
        riderorigin=(float(riderrow['Origin_Latitude']),float(riderrow['Origin_Longitude']))
        riderendpoint=(float(riderrow['Destination_Latitude']),float(riderrow['Destination_Longitude']))

        withmatchlength=geodesic(driverorigin, riderorigin).kilometers
        +geodesic(riderorigin, riderendpoint).kilometers+geodesic(riderendpoint, driverendpoint).kilometers
        matched3=matched3+1
        kstotal=kstotal+nomatchlength-withmatchlength
aks=kstotal/matched3
print(aks)
```

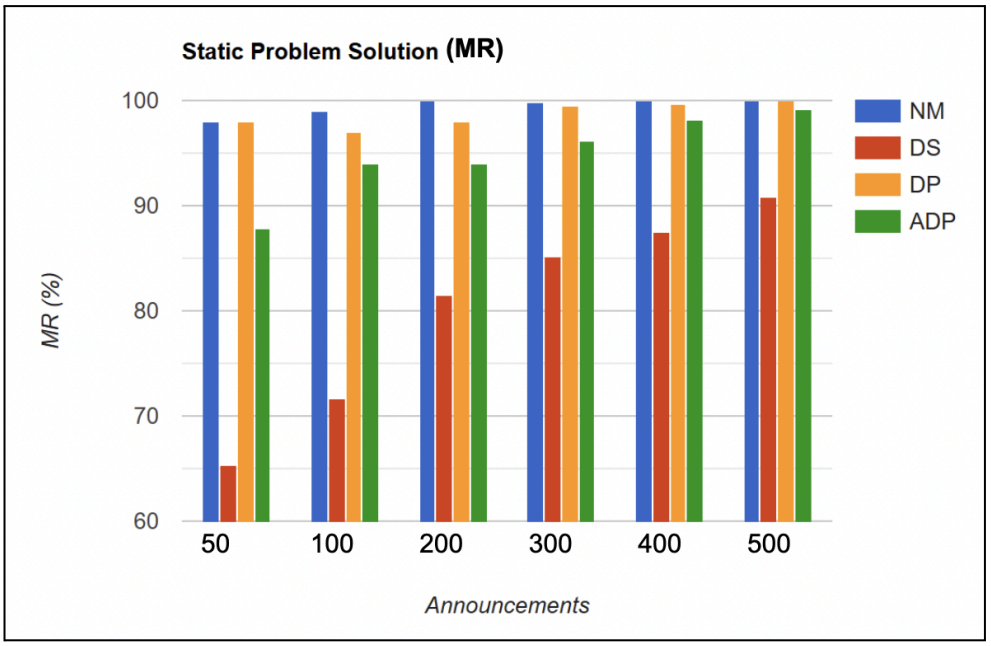
Figure 4.1.A: Code to run evaluation metrics on matched pairs

```
Driver 8134 is matched with Rider 100897
Driver 3716 is matched with Rider 106054
No Optimal Solution found
No Optimal Solution found
Driver 5720 is matched with Rider 102870
No Optimal Solution found
No Optimal Solution found
Driver 7871 is matched with Rider 107963
Driver 2981 is matched with Rider 102050
No Optimal Solution found
No Optimal Solution found
No Optimal Solution found
No Optimal Solution found
No Optimal Solution found
```

Figure 4.1.B: Sample solution found by model

The graph below illustrates the static problem solution in terms of the matching rate, which represents the percentage of successful pairings for the four weighting methods: number of matches, distance saved, distance proximity, and adjusted distance proximity. These matching rates are evaluated over sample sizes of 50, 100, 200, 300, 400, and 500 announcements for drivers and riders each.

Matching rates tend to be lower when there are fewer announcements. This is because a smaller pool of drivers and riders reduces the number of feasible matches. For variables such as distance saved or distance proximity, certain rider-driver pairs may be infeasible and are thus excluded. Conversely, as the number of announcements increases, the matching rates improve. A larger number of announcements generates a greater pool of potential driver-rider combinations, increasing the likelihood of feasible pairings.



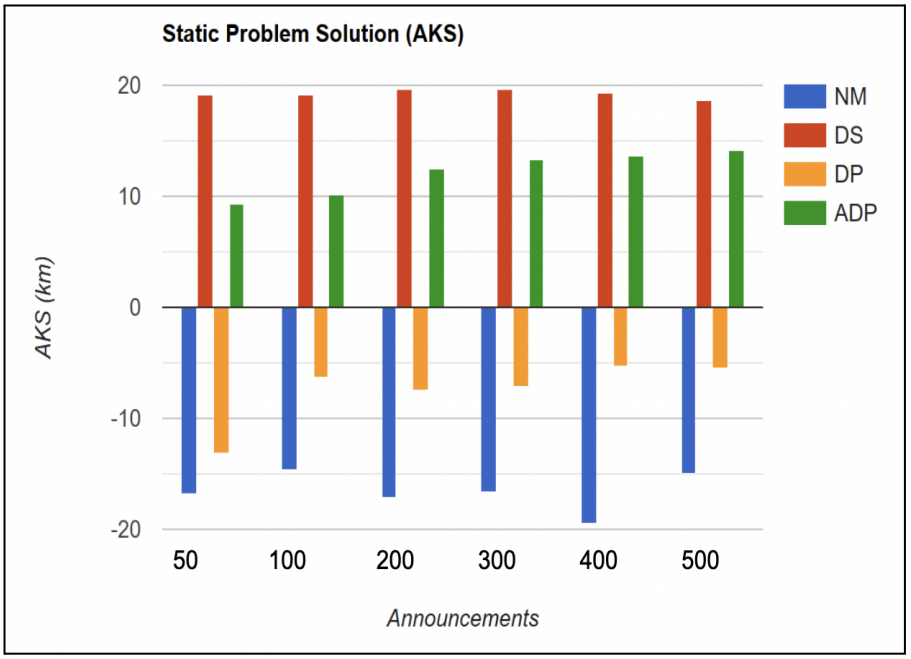
Method-Specific Observations

1. *Number of Matches:*
   The variable number of matches consistently achieves the highest matching rates across all announcement sizes. This is expected, as the requirement for this variable is relatively simple. It relies solely on the binary decision variable x_{dr}, which accounts only for whether a rider or driver submits a request to the system. This variable does not consider factors such as the distance between the rider and driver, the trip length, or, as in adjusted distance proximity, the proximity between the starting points of the rider and driver in addition to the trip length.
2. *Distance Saved:*
   The variable distance saved has the lowest matching rates across all announcement sizes. This is reasonable as if the shared distance is longer than the sum of the individual trips, then the coefficient to the binary variable for that pairing will be negative, and the optimization algorithm will often choose not to match the pair. At small sample sizes, there might even be specific drivers and riders that may not be matched with anyone, due to the unwillingness of the method to match pairs that don't have a positive distance saved. However, as the network gets bigger with a larger sample size, the matching rate goes up as network effects take hold.
3. *Distance Proximity/ Adjusted Distance Proximity:*
   Across all sample sizes, the adjusted distance proximity method exhibited lower matching rates compared to the unadjusted method. This outcome is expected, as the additional coefficient in each term of the summation expansion makes the algorithm more selective when determining feasible matches. However, as the sample size increases, the networking effects of a larger driver and rider pool result in the matching rates of both methods converging. We anticipated that as sample sizes approach one thousand, there will be

no functional difference in the matching rates between the adjusted and unadjusted distance proximity index weighting methods.

As the number of announcements increases, all four variables demonstrate an upward trend in matching rates. This convergence occurs because larger datasets provide a greater number of feasible driver-rider matches, allowing the functions to perform closer to their optimal levels.

As we can observe from the AKS performance chart below, as sample size increases, the Adjusted Distance Proximity (ADP) function is the only algorithm that shows consistent improvement in AKS performance, while the Distance Savings (DS) function remains relatively constant. This may be because the DS algorithm is more selective, producing fewer but highly optimal matches, whereas the ADP algorithm, which prioritizes relative proximity, benefits from networking effects in larger sample sizes. It can be hypothesized that as sample sizes grow further, the AKS performance of the ADP and DS algorithms may converge due to improved matching opportunities.



Method-Specific Observations

1. *Number of Matches:*
   This method demonstrates poor performance on the AKS metric, which is expected given that all pairings are assigned equal weights. As a result, the optimization algorithm lacks the ability to differentiate between efficient and inefficient matches, leading to consistently low AKS values.
2. *Distance Saved:*
   The Distance Saved (DS) method performs exceptionally well on the AKS metric, which is expected given its design. As a weighting method explicitly formulated to maximize kilometers saved, it achieves superior performance compared to other methods, aligning with its intended objective.
3. *Distance Proximity/ Adjusted Distance Proximity:*
   An interesting comparison can be made between the performance of Distance Proximity (DP) and Adjusted Distance Proximity (ADP). While both methods converged on the Matching Rate (MR) metric, with DP improving over time, DP performs significantly worse than ADP on the AKS metric. This discrepancy can be explained by their respective formulations. ADP considers the shared trip length, which accounts for the relative starting and ending locations of drivers and riders, whereas DP only considers the ratio of the

driver's and rider's individual trip lengths. Consequently, DP is more likely to match drivers and riders who are geographically distant, resulting in shared trips that are longer than the sum of their individual trips, leading to poor AKS performance.

4.2. Improvements

An alternative approach to solving this problem is the rolling horizon method, which addresses two key challenges: the high dynamicity of the system, where rider and driver requests continuously enter and leave, and the uncertainty of future requests, which are unknown at the time of optimization.

The rolling horizon methodology offers two primary optimization strategies:

1. *Periodic Optimization:* Operates at fixed time intervals, providing straightforward implementation but potentially slower response times.
2. *Event-Driven Optimization*: Reacts dynamically to events, offering faster response times but requiring more complex event definitions, making it highly effective in dynamic environments.

Additionally, the efficiency of the current code could be significantly improved. The current implementation struggles with large sample sizes, resulting in long runtimes, which limits scalability. A more optimized code would reduce runtime while enabling the analysis of much larger datasets, improving both computational efficiency and accuracy.

5. Appendix

We would like to clarify that AI tools were utilized solely to adjust the tone of the information presented; all the content is our original work. Additionally, AI tools were employed for the generation of specific graphs and models to enhance our analysis.

We sourced most of our methods and formulations from the research paper:
"Novel Dynamic Formulations for Real-Time Ride-Sharing Systems" by *Ali Najmi, David Rey, and Taha H. Rashidi (2017)*.

The implementation of the model was carried out using the following tools and libraries:

- **Anaconda** is used to manage the Python environment and dependencies.
- **Jupyter Notebook** for interactive coding and analysis
- **gurobi.py** for optimization modeling and solving
- **geo.py** for geographical data handling and processing