Servicios de movilidad on-demand en tiempo real

Diseño de Algoritmos - Licenciatura en Tecnología Digital - UTDT

Luca Mazzarello* Ignacio Pardo†

2023-06-02

Abstract

El problema a investigar consiste en la asignación de vehículos disponibles a pasajeros que demanden de un viaje. Este trabajo presenta tres algoritmos con distintos enfoques para el problema de asignación de taxis. Una heurística golosa que priorice los pasajeros por orden de llegada, un algoritmo que minimice las distancias recorridas por los taxis globales y un algoritmo que minimice los viajes menos rentables para los taxistas. El objetivo de este informe es el de detallar los algoritmos y compararlos en términos de performance y calidad de solución.

Contents

UNIVERSIDAD TORCUATO DI TELLA

*lmazzarello@mail.utdt.edu
†ipardo@mail.utdt.edu

1 Introducción

Nuestro objetivo principal en este estudio es analizar y evaluar el proceso de asignación de vehículos a pasajeros, considerando principalmente la distancia entre ellos y el recorrido de los viajes a realizar. En el rol de consultores para una empresa de servicios de movilidad, nos enfocamos en actuar como intermediarios entre pasajeros y vehículos, ya sean taxis o vehículos privados.

1.1 Motivación

En el mercado local, existen diversas aplicaciones que facilitan la conexión entre pasajeros y vehículos, pero es crucial comprender cómo mejorar y optimizar el proceso de asignación para garantizar una experiencia óptima tanto para los pasajeros como para los conductores. En este sentido, nos centraremos en la toma de decisiones en tiempo real, utilizando datos relevantes como la distancia del viaje y la ubicación actual del vehículo para realizar una asignación eficiente y conveniente.

Al considerar dichas cuestiones, buscamos encontrar soluciones que minimicen los tiempos de espera de los pasajeros, optimicen la eficiencia de los vehículos y mejoren la calidad general del servicio. Nuestro análisis se basará en evaluar diferentes enfoques y algoritmos de asignación.

1.2 Consigna y descripción del problema

En un instante dado (o en un intervalo muy pequeño de tiempo, digamos 10-15 segundos) y en una determinada área geográfica la empresa posee un número de pedidos de viajes a realizar y una cierta cantidad de vehículos disponibles para realizar viajes. Para cada pasajero podemos asumir la disponibilidad de un mínimo de información como

- el instante en el que realizó el pedido;
- la localización de origen del viaje (donde debe ser buscado);
- la localización de destino del viaje (donde debe ser llevado) o la distancia estimada del recorrido;
- una estimación de la tarifa total a cobrar por el viaje (que depende en parte de la distancia del recorrido, pero puede contener gastos extra como valijas, cantidad de pasajeros, horas pico, etc.).

Análogamente, también podemos asumir mínimamente conocer en tiempo real la localización de cada uno de los vehículos de nuestra flota, en particular la de aquellos que estén disponibles para realizar viajes. Combinando la localización de pasajeros y vehículos, podemos asumir también que conocemos la distancia a recorrer que le llevaría a cada vehículo llegar a cada posible cliente a fin de poder empezar el viaje. El problema que buscamos resolver es decidir que vehículo debe buscar a cada pasajero.

A fin de formular un modelo para la decisión en cuestión, primero formalizamos el problema. Tenemos n vehículos disponibles, $i = 1, \ldots, n$ para cubrir n viajes de distintos pasajeros, $j = 1, \ldots, n$. Por simplicidad, asumimos que el problema se encuentra balanceado en términos de oferta y demanda de viajes, es decir, contamos con la misma cantidad de pasajeros demandantes que de vehiculos disponibles. En función de la información geográfica de los conductores y pasajeros, definimos d_{ij} a la distancia que debe recorrer el conductor i para empezar el viaje del pasajero j. Adicionalmente, para un pasajero $j = 1, \ldots, n$ llamamos v_j a la distancia del viaje a realizar por el pasajero j y f_j a la tarifa total a cobrarle por el viaje. Asumimos también que los pasajeros se encuentran ya ordenados de manera creciente en función del instante en el que realizaron el pedido.

1.2.1 Estrategias de asignación

Como indíca la consigna: La decisión puede ser abordada con distintos enfoques. Una primera aproximación natural al problema consiste en atender a los pasajeros uno a uno siguiendo el criterio First Come, First Served (FCFS), y tomar para cada pasajero una decisión **greedy** asignando el conductor disponible más cercano. En base a nuestras definiciones, esta estrategia consiste en los siguientes pasos:

- considerar a los pasajeros por orden de llegada;
- a cada pasajero asignar el vehículo más cercano disponible.
- cada vehículo debe ser usado por exactamente un pasajero.

Esta corresponde a la estrategia aplicada actualmente por la empresa.

Sin embargo, aún con un contexto de toma de decisión en tiempo real, si el problema presenta un volumen de demanda significativo es posible modificar el proceso aplicando la idea de matching si se esperan algunos segundos y se arma un batch.

Para ampliar y mejorar la resolución del problema, se puede observar la estrategia de Uber que consiste en agrupar los viajes en batches de manera tal que se busca minimizar la distancia **total** recorrida por los vehículos.

Es posible modificar el proceso aplicando la idea de matching si se esperan algunos segundos y se arma un batch. El concepto detrás de esta idea es simple: dentro de los parámetros permitidos, tratar de agrupar varios pedidos y, en lugar de tomar decisiones locales a cada pasajero, reformular el problema de manera global y tomar una decisión conjunta.

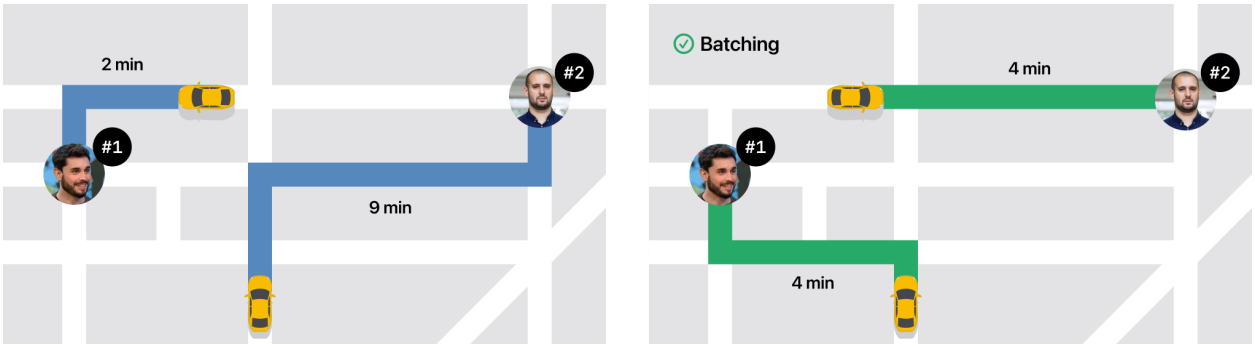


Figure 1: Comparación Asignaciónes - Fuente: Uber

En nuestro caso, podemos asumir que los n vehículos disponibles y los n pasajeros solicitan un viaje en un lapso de tiempo pequeño, algunos pocos segundos, en los cuales el correspondiente usuario está esperando una repuesta (ya sea pasajero, con su vehículo asignado, o un vehículo, con su pasajero). Luego, el objetivo es formular un modelo que tome una decisión global, indicando qué vehículo es asignado a cada pasajero.

Inicialmente, consideramos como métrica de éxito la minimización de la distancia recorrida por los vehículos hasta la ubicación de su pasajero asignado. Dado que asumimos la oferta y demanda balanceada, todos los pedidos deben ser cumplidos. Nuestro objetivo es proveer evidencia basándonos en datos y metodología formal respecto a la mejora en la distancia total recorrida por los vehículos. Esta métrica se puede utilizar como un proxy de los costos y tiempos de espera de los pasajeros, bajo la hipótesis que la distancia tiene algún tipo de correlación con estos otros factores.

Luego, se buscará plantear otro modelo que, en respuesta a la necesidad de los conductores de realizar menos tiempo viajes de buscar pasajeros que de transportarlos, busque minimizar dicha métrica sin perder de vista la distancia total recorrida por los vehículos.

2 Resolución del problema

Contamos con un desarrollo básico nuestro problema, incluyendo una clase TaxiAssignmentInstance que se encarga de leer la definición de una instancia del problema de un archivo de entrada. La misma contiene los siguientes atributos:

- `n`: cantidad de vehículos/pasajeros.
- `taxis_position: vector<pair(double,double)>` de n elementos, donde la posición i tiene la localización en términos de longitud y latitud del taxi $i, i = 1, \ldots, n$.
- `paxs_position: vector<pair(double,double)>` de n elementos, donde la posición j tiene la localización en términos de longitud y latitud del pasajero $j, j = 1, \ldots, n$.
- `paxs_trip_dist: vector<double>`, donde la posición j tiene la distancia (en kms) del viaje a realizar por el pasajero $j, j = 1, \ldots, n$.
- `paxs_tot_fare: vector<double>`, donde la posición j tiene la tarifa total (en USD) del viaje a realizar por el pasajero $j, j = 1, \ldots, n$.
- `dist: vector<vector<double>>` con una matriz que en la posición (i, j) posee la distancia d_ij (en kms) que debe recorrer el vehículo i para empezar el viaje del pasajero $j, i, j = 1, \ldots, n$. Notar que los pasajeros están representados por las columnas $(j = 1, \ldots, n)$.

Para la evaluación del problema, se cuenta con 4 sets de instancias de distintos tamaño, para modelar escenarios variados de demanda. Las mismas deben ser utilizadas para comparar la efectividad de nuestro enfoque y realizar una comparación extensiva entre el método de FCFS y el de matching. Las características de las instancias son:

- `small`: $n = 10$;
- `medium`: $n = 100$;
- `large`: $n = 250$;
- `xl`: $n = 500$.

Cada grupo posee 10 instancias distintas, a fin de agregar variabilidad en los escenarios considerados.

A partir de estas instancias, se obtiene un conjunto discreto de resultados de tamaño n, por lo que para expandir y generalizar la evaluación de los resultados, se generaron también, para cada $n \in [3, 500]$, 10 instancias con datos aleatorios de las mismas características que las instancias de los sets de datos originales. Estas instancias se utilizaron para evaluar el comportamiento de los algoritmos a medida que aumenta la cantidad de vehículos y pasajeros de forma mas general.

2.1 Algoritmo 1: Greedy Solver (FCFS)

Como plantea la consigna, el primer approach a la asignación de taxis a pasajeros se moldea a partir de una estrategia FCFS (First Come, First Served). En este caso, se considera a los pasajeros por orden de llegada y a cada pasajero se le asigna el vehículo más cercano.

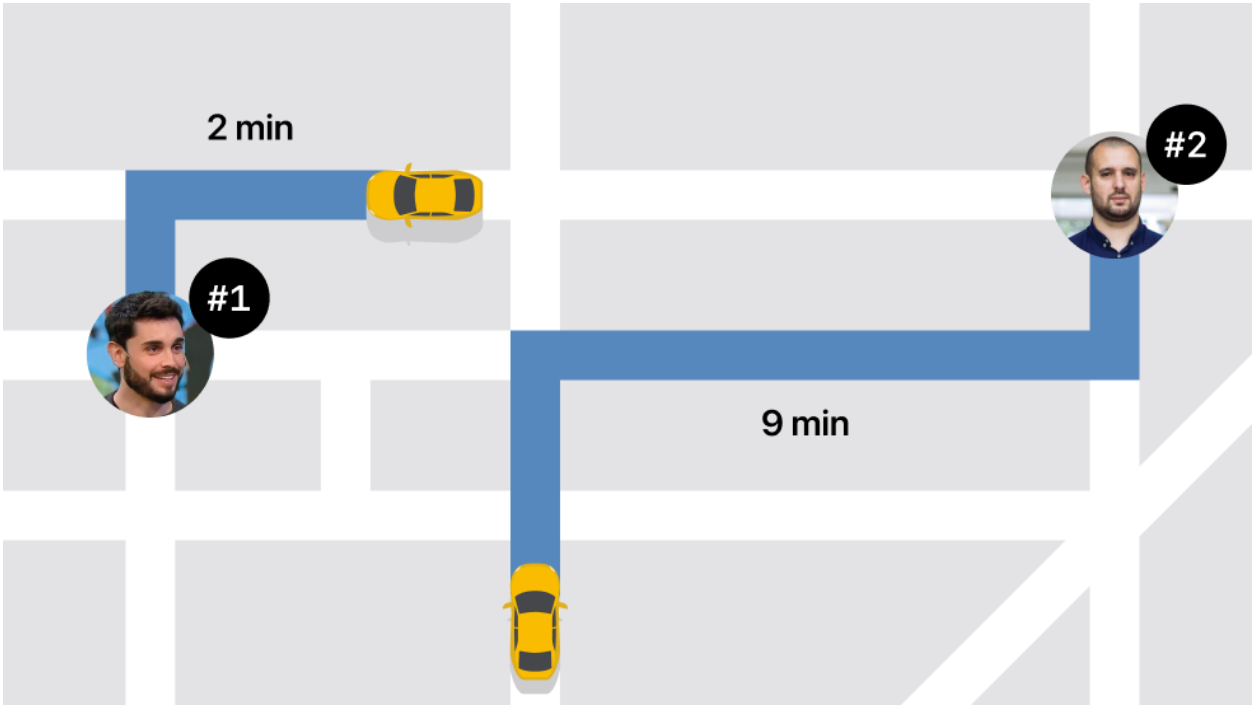


Figure 2: Greedy Solver

2.1.1 Implementación FCFS

El algoritmo comienza creando una instancia de la clase `GreedySolver` y guarda la instancia del problema de asignación de taxis y su valor objetivo en variables internas. También inicializa el estado de la solución y el tiempo de ejecución.

El método `solve()` es el corazón del algoritmo. Comienza inicializando una solución vacía, representada por el objeto `TaxiAssignmentSolution`. Luego, itera sobre cada pasajero en orden ascendente. Para cada pasajero, se busca el taxi más cercano que **aún no haya sido asignado** a otro pasajero. Esto se realiza mediante un bucle que busca el primer taxi disponible. Una vez encontrado, se asigna ese taxi como el mínimo temporal y se procede a comparar la distancia entre el taxi mínimo y el pasajero actual con la distancia de otros taxis no asignados. Si se encuentra un taxi más cercano, se actualiza el taxi mínimo y su distancia.

Después de asignar todos los pasajeros, se registra el tiempo de ejecución, el valor objetivo y el estado de la solución. El estado de la solución es análogo al descripto en la documentación de `OR-TOOLS`.

```
enum Status {
    NOT_SOLVED,
    OPTIMAL,
    FEASIBLE,
    INFEASIBLE,
    UNBALANCED,
    BAD_RESULT,
    BAD_COST_RANGE
};
```

2.2 Algoritmo 2: Batching Matching

La idea detrás de una estrategia de batching es agrupar los viajes de los pasajeros en grupos de forma tal que se minimice la distancia total recorrida por los taxis. Para esto, se plantea un modelo de optimización que resuelve el problema de encontrar el agrupamiento óptimo de los viajes de los pasajeros.

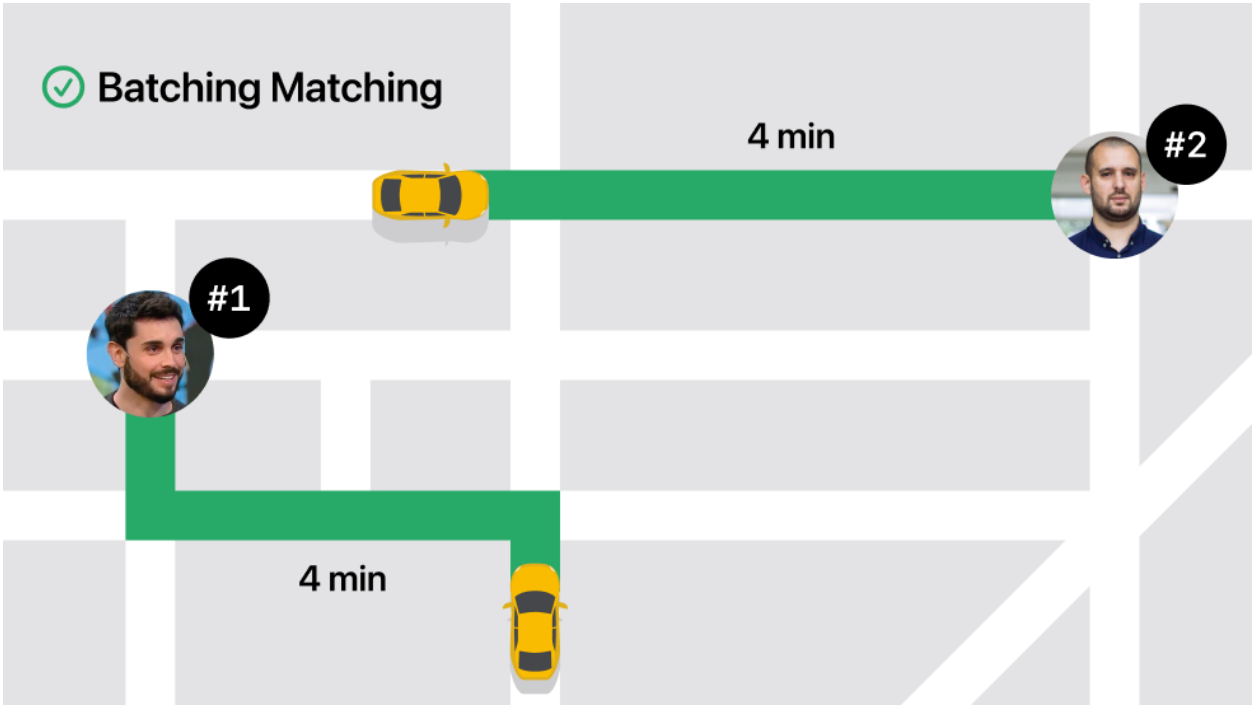


Figure 3: Batching Matching

2.2.1 Modelo para estrategia de batching

Para la estrategia de batching, se plantea una red de flujo de costo mínimo, donde los nodos representan los taxis y los pasajeros, y las aristas representan los viajes que se pueden realizar. Cada arista tiene un costo asociado, que es la distancia que debe recorrer el taxi para llegar al pasajero. Además, cada arista tiene una capacidad asociada, que es la cantidad de viajes que puede realizar el taxi. La capacidad de las aristas que conectan los taxis con los pasajeros es 1, ya que cada taxi solo puede realizar un viaje. La capacidad de las aristas que conectan los pasajeros con los taxis es la cantidad de taxis disponibles, ya que cada pasajero puede ser asignado a cualquier taxi. Por último, se agrega un nodo fuente (s) y un nodo sumidero (t), que representan los viajes que se pueden realizar. El nodo fuente tiene una arista que lo conecta con cada taxi, con capacidad 1 y costo 0. El nodo sumidero tiene una arista que lo conecta con cada pasajero, con capacidad 1 y costo 0. El objetivo es encontrar el flujo de costo mínimo que maximice la cantidad de viajes realizados.

El grafo resultante es $G = (V, E)$ donde V es el conjunto de nodos y E es el conjunto de aristas.

Los nodos se dividen en 3 conjuntos:

$$V = \{V_{taxis}, V_{pasajeros}, V_{fuente-sumidero}\}$$

V_{taxis} es el conjunto de nodos que representan a los taxis, $V_{pasajeros}$ es el conjunto de nodos que representan a los pasajeros y $V_{fuente-sumidero}$ es el conjunto de nodos que representan a la fuente y al sumidero. Los nodos de V_{taxis} y $V_{pasajeros}$ se definen de la siguiente forma:

$$V_{taxis} = \{t_1, t_2, ..., t_m\}$$
$$V_{pasajeros} = \{p_1, p_2, ..., p_n\}$$

Luego, siendo d_{ij} la distancia entre el taxi t_i y el pasajero p_j, el conjunto de aristas E se define de la siguiente forma:

$$E = (t_i, p_j, d_{ij}, 1) \forall t_i \in V_{taxis}, p_j \in V_{pasajeros}$$

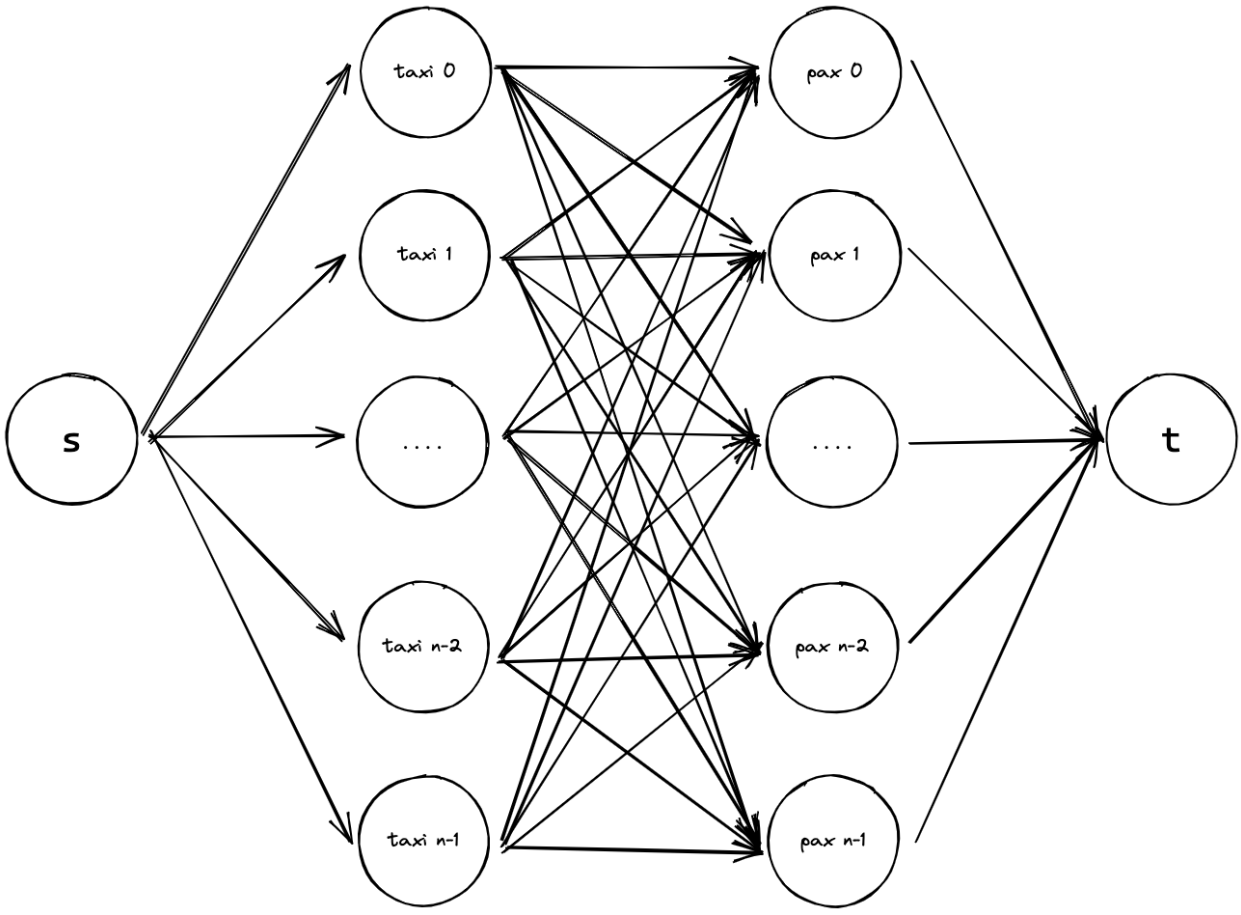


Figure 4: Batching Matching Model

2.2.2 Implementación Batching

Para implementar el modelo de batching, se utilizó la librería `OR-TOOLS`. Esta librería permite resolver problemas de optimización combinatoria, como el problema de flujo de costo mínimo. Para resolver el problema, se encapsuló la lógica de la librería en una clase llamada `BatchingSolver` que implementa el método `solve()`.

La clase BatchingSolver tiene dos constructores: uno sin argumentos y otro que recibe una instancia de `TaxiAssignmentInstance`. Estos constructores inicializan los atributos de la clase, como el valor objetivo, el estado de la solución y el tiempo de solución. El método `setInstance()` se utiliza para establecer la instancia de `TaxiAssignmentInstance` en el atributo `_instance` de la clase.

El método `solve()` se encarga de resolver el problema de asignación de taxis utilizando un grafo de flujo de costo mínimo. Para ello, se lleva a cabo el proceso de creación de la red de flujo de costo mínimo por parte del método `_createMinCostFlowNetwork()` que se resuelve el problema utilizando el método `Solve()` del objeto `MinCostFlow` de la biblioteca `OR-TOOLS`. Luego, si la solución es óptima, se asignan los taxis a los pasajeros correspondientes y se calcula el costo de la solución.

El método `_createMinCostFlowNetwork()` se encarga de crear la red de flujo de costo mínimo para el problema de asignación de taxis. Esto implica la definición de los nodos, las capacidades, los costos unitarios y los suministros de la red. Se establecen los arcos entre los taxis y los pasajeros, y se asignan los suministros correspondientes a los nodos de los taxis y los pasajeros.

Primero, se obtiene el número de taxis y pasajeros de la instancia guardada en el atributo `_instance`. Luego se crean varios vectores que almacenarán la información de la red, como los nodos de inicio, los nodos de fin, las capacidades y los costos unitarios.

El siguiente paso es llenar los vectores `start_nodes`, `end_nodes` y `unit_costs` con la información necesaria para crear los arcos de la red. Los vértices de la red se indexan de 0 a $2n-1$, donde los primeros n representan a los taxis y los últimos n representan a los pasajeros.

Se recorren todos los taxis y pasajeros y se establecen las conexiones entre ellos. Cada conexión representa un arco en la red de flujo de costo mínimo. Los arcos tienen una capacidad de 1 (ya que cada taxi solo puede atender a un pasajero) y un costo unitario que se calcula como 10 veces la distancia entre el taxi y el pasajero. El factor de escala de 10 se utiliza para que los costos sean enteros y no de tipo `double`, manteniendo cierta precision, ya que la librería `OR-TOOLS` solo acepta costos enteros para los arcos.

$$\text{UnitCost}(t_i, p_j) = \lfloor 10 \times \text{Distance}(t_i, p_j) \rfloor \quad \forall t_i \in T, p_j \in P$$

Después de llenar los vectores con la información de los arcos, se crea la red de flujo de costo mínimo en el atributo `_min_cost_flow` utilizando el objeto `MinCostFlow`. Se agrega cada arco a la red utilizando el método `AddArcWithCapacityAndUnitCost()`.

Luego se establecen los suministros de cada nodo en la red utilizando el método `SetNodeSupply()`. Los taxis tienen un suministro de 1, lo que indica que pueden atender a un pasajero, mientras que los pasajeros tienen un suministro de -1, lo que indica que necesitan ser atendidos por un taxi.

Una vez que se han creado todos los arcos y se han establecido los suministros de los nodos, la red de flujo de costo mínimo está lista para ser utilizada en la resolución del problema.

Al implementar nuestro modelo de batching, los nodos de fuente y sumidero no fueron necesarios ya que a diferencia de los grafos a los que estábamos acostumbrados a trabajar con caminos, la red de flujo de costo mínimo no requiere de un nodo de inicio y un nodo de fin.

2.3 Herramientas utilizadas

2.3.1 Chrono C++

Para cronometrar el tiempo de ejecución de los algoritmos, se utilizó la librería `chrono` de C++. Esta librería permite medir el tiempo de ejecución de un programa en nano-segundos. La forma en la que se utilizó `chrono` es la siguiente:

```
// Inicio del cronómetro
auto start = std::chrono::steady_clock::now();
* * * Código a cronometrar * * *
// Fin del cronómetro
auto time = std::chrono::duration<double, std::milli>(end - start).count();
```

2.3.2 Google `OR-TOOLS`

Para implementar el modelo de `Batching`, se utilizó la librería `OR-TOOLS`. Esta librería permite resolver problemas de optimización combinatoria, como el problema de flujo de costo mínimo. Para resolver el problema de Batching Matching se utilizo la clase `MinCostFlow` de la librería. Esta clase representa un grafo de flujo de costo mínimo. Para crear la red de flujo de costo mínimo, se utilizó el método `AddArcWithCapacityAndUnitCost()` de la clase `MinCostFlow`. Este método se encarga de agregar un arco a la red de flujo de costo mínimo. El método `Solve()` de la clase `MinCostFlow` se encarga de resolver el problema de flujo de costo mínimo.

2.3.3 Jupyter Notebook

Para realizar el análisis de los resultados, se utilizó un notebook de Jupyter. Este notebook se encuentra en el archivo `src/scripts/experimentacion.ipynb`. Este notebook se encarga de leer los resultados de las corridas de los algoritmos y de calcular el valor objetivo de cada corrida, para luego generar los gráficos correspondientes. Ademas, todas las instancias aleatorias se generaron utilizando este notebook.

3 Resultados

3.1 Obtención de resultados

Para obtener los resultados, se corrieron los dos algoritmos para cada instancia de cada set de datos y se guardaron los resultados de cada corrida. Luego, se calculó el valor objetivo de cada corrida y se guardaron en el archivo `src/output/results.csv`.

La siguiente tabla detalla un resumen de los resultados obtenidos:

	n	greedy_cost	batching_cost	greedy_time	batching_time
count	40	40	40	40	40
mean	215	696.783	587.278	0.442864	23.525
std	187.903	583.801	498.485	0.538287	30.6385
min	10	29.5	23.7	0.001958	0
25%	77.5	219.875	184.925	0.046667	1.5
50%	175	572.8	471.85	0.207666	8.5
75%	312.5	1088.7	912.875	0.607448	34
max	500	1737.4	1521.8	1.43354	82

A simple vista podemos ver que la media del valor objetivo de la solución obtenida por el algoritmo de `Batching` es menor que la del algoritmo `Greedy` (587.278 vs 696.783). Esto indica que el algoritmo de `Batching` obtiene mejores resultados que el algoritmo greedy. Sin embargo, el tiempo de ejecución del algoritmo de `Batching` es mucho mayor que el del algoritmo `Greedy`.

3.2 Análisis de resultados

3.2.1 Distancia total recorrida

Para comparar los resultados obtenidos por los algoritmos, se utiliza el valor objetivo de sus soluciones. En cada caso, se mide la mejora porcentual obtenida entre soluciones. Sean z_b y z_g el valor de la función objetivo de una solución del modelo para el batching y el de FCFS, respectivamente. Definimos la mejora relativa y aplicamos la formula a los valores medidos:

$$\%\text{gap} = \frac{z_g - z_b}{z_b}$$

n	greedy_cost	batching_cost	gap
10	46.93	40.06	0.177322
100	337.31	280.63	0.205883
250	849.6	701.6	0.212157
500	1553.29	1326.82	0.171215

A simple vista parece ser que el algoritmo de batching obtiene mejores resultados que el algoritmo greedy, ya que la mejora relativa es positiva en todos los casos. Buscaremos observar entonces la progresión de la mejora relativa a medida que aumenta la cantidad de pasajeros. Generamos una regresión cuadrática y una logarítmica para observar la tendencia de la mejora relativa a medida que aumenta la cantidad de pasajeros. Para obtener una mejor idea de la distribución de los datos, se graficó un histograma de la mejora relativa entre algoritmos así también como sus distribuciones para cada n pasajeros.

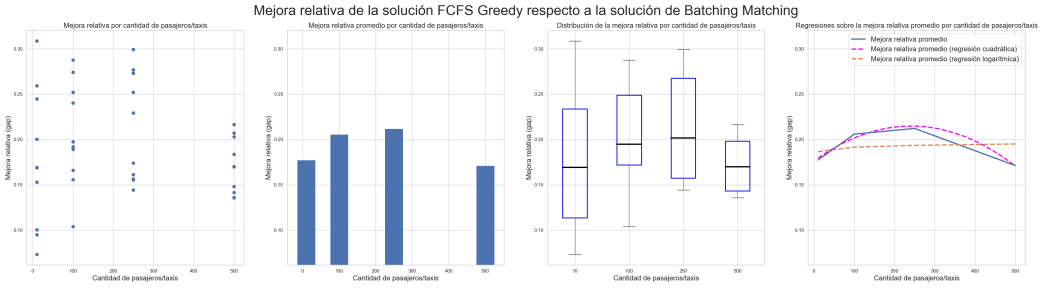


Figure 5: Greedy vs Batching Cost

n	count	mean	std	min	25%	50%	75%	max
10	10	0.177322	0.0769256	0.0732394	0.113712	0.169066	0.233651	0.308642
100	10	0.205883	0.0574211	0.104009	0.171945	0.194773	0.249033	0.287764
250	10	0.212157	0.0599257	0.144362	0.157622	0.201565	0.267717	0.299222
500	10	0.171215	0.0305612	0.135713	0.143316	0.170003	0.198171	0.216486

Es en este gráfico se puede ver que el desvío estándar de las mejoras relativas decrece a medida que aumenta la cantidad de pasajeros. Esto indica que la mejora relativa del algoritmo de `Batching` por sobre el algoritmo `Greedy` se vuelve mas consistente a mayor cantidad de pasajeros. Ademas parece ser que la media de las mejoras relativas disminuye a medida que aumenta la cantidad de pasajeros. Para expandir el análisis, se realizó la misma evaluación sobre las ~5000 instancias generadas aleatoriamente.

3.2.2 Instancias aleatorias

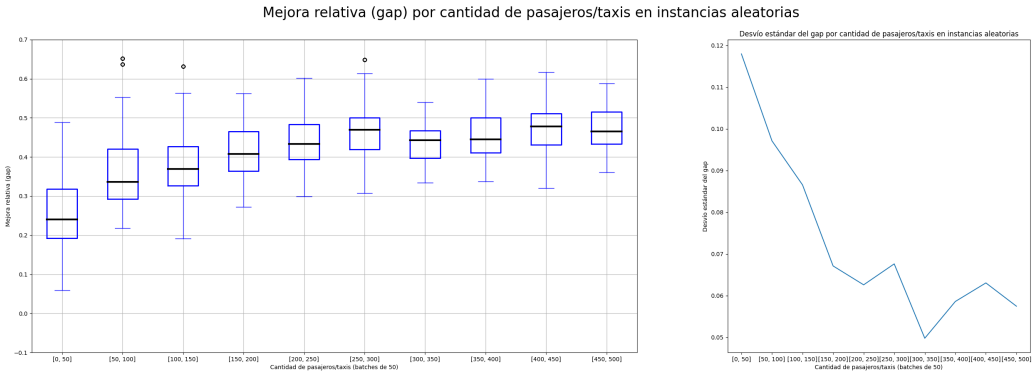
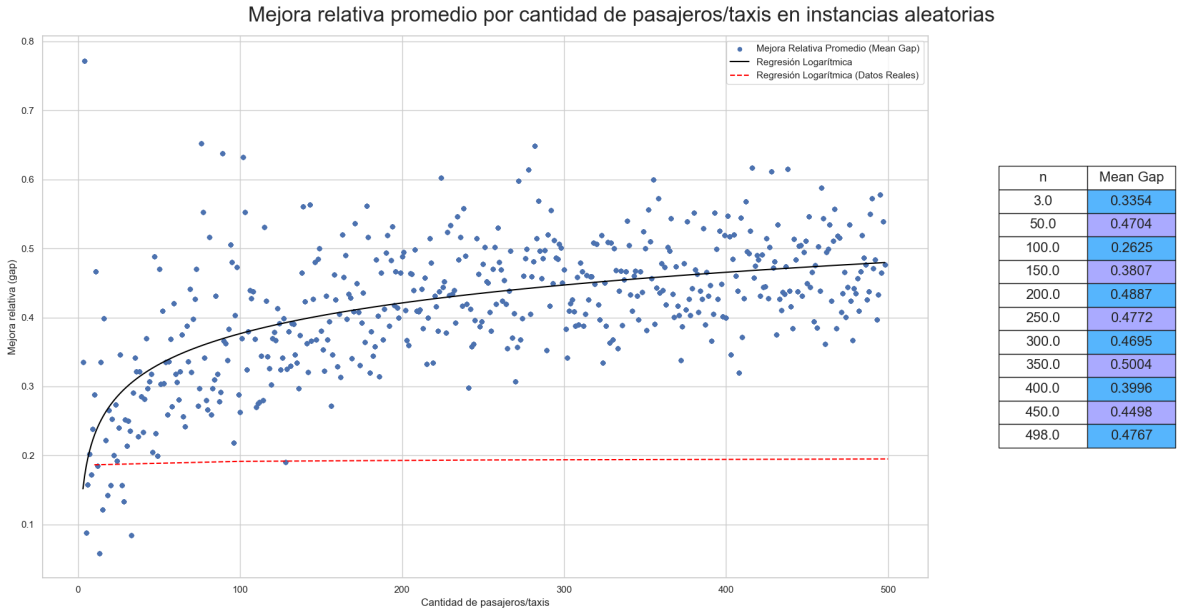


Figure 6: Mejora Relativa - Greedy vs Batching Cost - std - Random Instances

Nuevamente se observa que el desvío estándar de las mejoras relativas decrece a medida que aumenta la cantidad de pasajeros. Para observar la tendencia de la mejora relativa a medida que aumenta la cantidad de pasajeros, se realizó una regresión logarítmica sobre la media de las mejoras relativas para cada cantidad de pasajeros.



n	Mean Gap
3.0	0.3354
50.0	0.4704
100.0	0.2625
150.0	0.3807
200.0	0.4887
250.0	0.4772
300.0	0.4695
350.0	0.5004
400.0	0.3996
450.0	0.4498
498.0	0.4767

Figure 7: Mejora Relativa - Greedy vs Batching Cost Log Regression - Random Instances

Al Scatter Plot lo acompaña una regresión logarítmica a partir de la media de las mejoras relativas para cada cantidad de pasajeros para poder observar la tendencia de la mejora relativa a medida que aumenta la cantidad de pasajeros. A partir de estos resultados, interpretamos que, a medida que la cantidad de pasajeros crece, la mejora relativa de la solución obtenida por el algoritmo de batching respecto a la solución obtenida por el algoritmo greedy tiende a crecer cada vez mas lento. El valor de la mejora relativa promedio tiende de forma logarítmica en un valor cercano a 0.5 cuando $n = 500$, mientras que para las 40 instancias originales la mejora relativa promedio tiende a 0.2, lo cual indica que el algoritmo de `Batching` es aún mejor por sobre el de `Greedy` que lo observado en la sección anterior.

3.2.3 Tiempo de ejecución

La complejidad temporal del algoritmo greedy es $O(n^2)$, ya que en cada iteración se debe calcular la distancia entre cada par de pasajeros. La complejidad del algoritmo de batching es $O(n^2 * m * log(n * C))$ por la complejidad de resolver el flujo máximo de costo mínimo que esta descripta en la documentación de la clase `MinCostFlow` de `OR-TOOLS` a partir del algoritmo propuesto en *Finding Minimum-Cost Circulations by Successive Approximation* de Goldberg y Tarjan.

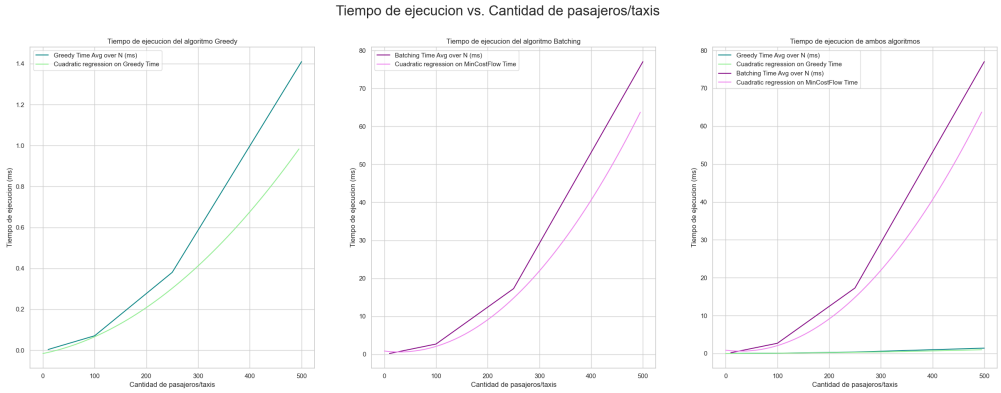


Figure 8: Tiempo de Ejecución - Greedy vs Batching - Instancias Originales

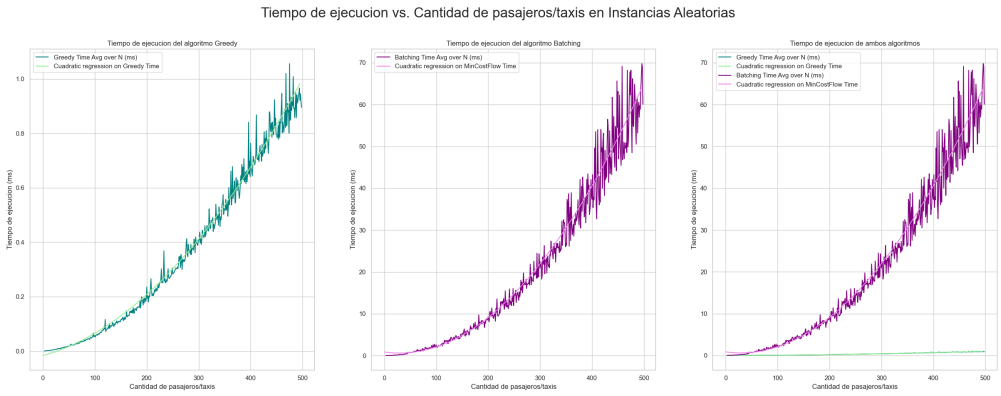


Figure 9: Tiempo de Ejecución - Greedy vs Batching - Instancias Aleatorias

De modo empírico, se observó que el algoritmo greedy es mucho más rápido que el algoritmo de batching. Obtenemos para el dominio $n = 1, \ldots, 500$, la imagen del tiempo de ejecución del algoritmo greedy esta entre 0.001 y 2.66279 mili-segundos, mientras que la imagen del tiempo de ejecución del algoritmo de batching esta entre 0.0 y 151.0 mili-segundos. En el gráfico el tiempo de ejecución es el del promedio de las 10 ejecuciones de cada algoritmo para cada cantidad de pasajeros, las imágenes para estos resultados se encuentran entre 0 y ~1 ms para el algoritmo greedy y entre 0 y ~70 ms para el algoritmo de batching.

Todos los tiempos de ejecución fueron medidos en una computadora Macbook Pro M1 2020 con 16GB de RAM y 8 núcleos de procesamiento, corriendo macOS Ventura 13.3.1 (a) (22E772610a). El compilador utilizado fue g++ v. c++17.

4 Un nuevo modelo: Taxi Priority

4.1 Motivación

En base a encuestas realizadas a los conductores en general, las aplicaciones sugieren muchas veces viajes que demandan una distancia considerable para llegar a la ubicación del pasajero, para luego realizar un muy viaje corto en comparación. En este sentido, la sensación de los conductores es que es mucho el costo de buscar el pasajero, ya sea en costo específico o en el tiempo utilizado y que podrían destinar a un viaje más rentable, en relación al beneficio obtenido por el viaje en sí mismo.

Para mitigar este problema, se planteo un nuevo algoritmo para la asignación de Taxis a Pasajeros teniendo en cuenta:

- La distancia entre el Taxi y el Pasajero
- La distancia del viaje del Pasajero

De esta forma, nuestra nueva métrica a optimizar es una función de la distancia del viaje y la distancia del Taxi al Pasajero resultando en un algoritmo *Taxi Centered*, es decir que prioriza los viajes más rentables para los conductores.

4.2 Modelo y Algoritmo `TaxiPrioritySolver`

Como la cuestión a resolver es minimizar el costo total de asignar los Taxis a los Pasajeros, se define la función de costo unitario como la distancia entre el Taxi y el Pasajero dividido la distancia del viaje del Pasajero. De esta forma, se busca minimizar el costo unitario de asignar un Taxi a un Pasajero. Por ejemplo, si tenemos un Taxi t_i y un Pasajero p_j donde:

- La distancia entre el Taxi y el Pasajero es $\text{Distance}(t_i, p_i) = 10$
- La distancia del viaje del Pasajero es $\text{Distance}(p_i) = 20$

El costo unitario de asignar el Taxi al Pasajero es $\frac{\text{Distance}(t_i,p_i)}{\text{Distance}(p_i)} = \frac{10}{20} = 0.5$.

A modo de implementación, optamos por multiplicar el costo unitario por 100, para poder trabajar con números enteros, pero a su vez, mantener cierta precisión al estar hablando de un ratio de distancias. En el caso de que el pasajero realize un viaje de distancia = 0, el costo del arco pasa a valer ∞. La librería `limits` de `C++` incluye una definición para un `INT_MAX`: `numeric_limits<int>::max()`, tal que `INT_MAX` $> i \forall i \in Z$. De esta forma, el costo unitario queda definido como:

$$\text{TaxiPriority Cost}(t_i, p_j) = \lfloor 100 * \frac{\text{Distance}(t_i, p_j)}{\text{Distance}(p_j)} \rfloor \quad \forall t_i \in T, p_j \in P$$

La relación entre la distancia del Taxi al Pasajero y el costo unitario es directamente proporcional, es decir que a mayor distancia, mayor costo unitario. Por otro lado, la relación entre la distancia del viaje del Pasajero y el costo unitario es inversamente proporcional, es decir que a mayor distancia, menor costo unitario. De esta forma, buscamos minimizar el costo unitario, es decir que se priorizan los Taxis más cercanos a los Pasajeros que tienen viajes más largos. Llamemos entonces al total de costos unitarios de asignar los Taxis a los Pasajeros para una solución como "Costo de Taxistas".

Un detalle a tener en cuenta es que la función de costo unitario del modelo de `Batching` es directamente proporcional con la función de costo unitario de `TaxiPriority` ya que ambas dependen directamente de la distancia entre el Taxi y el Pasajero de la siguiente forma:

$$\text{Batching UnitCost}(t_i, p_j) = \lfloor 10 \times \text{Distance}(t_i, p_j) \rfloor \quad \forall t_i \in T, p_j \in P$$

Este problema lo podemos modelar nuevamente como un problema de flujo de costo mínimo, donde los nodos son los Taxis, los Pasajeros y el nodo fuente y sumidero. Las aristas son las conexiones entre los Taxis y los Pasajeros, pero en este caso el costo de cada arista es el nuevo costo unitario de asignar el Taxi al Pasajero. La capacidad de cada arista es 1, ya que cada Taxi solo puede ser asignado a un Pasajero.

4.3 Resultados

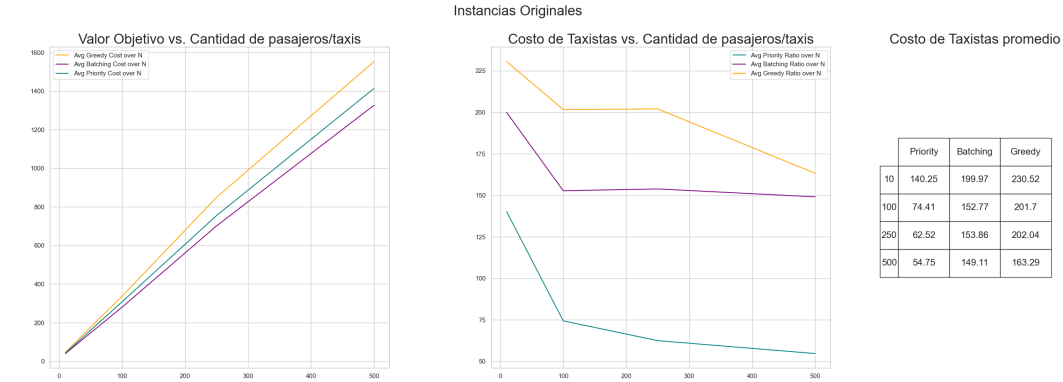
En la experimentación se plantearon las siguientes cuestiones a comparar a través de los modelos.

- Los costos objetivos de las soluciones obtenidas por cada modelo. (Batching Cost)
- La nueva métrica de Costo de Taxistas. (TaxiPriority Cost)
- Los tiempos de ejecución de cada modelo.

Al momento de calcular el Costo de Taxistas promedio, aquellos costos que en la implementación fueron definidos como `INT_MAX` fueron descartados ya que, como dependen de las distancias de viajes de los pasajeros por estar dados por las instancias y no por las soluciónes generadas, no aportan información distinctiva a los promedio de los costos a traves de los tipos de soluciones.

4.3.1 Comparación de soluciones en Instancias Originales

Inicialmente evaluamos los tres modelos en las instancias originales.



	Priority	Batching	Greedy
10	140.25	199.97	230.52
100	74.41	152.77	201.7
250	62.52	153.86	202.04
500	54.75	149.11	163.29

Figure 10: TaxiPriority vs Batching vs Greedy - Instancias Originales

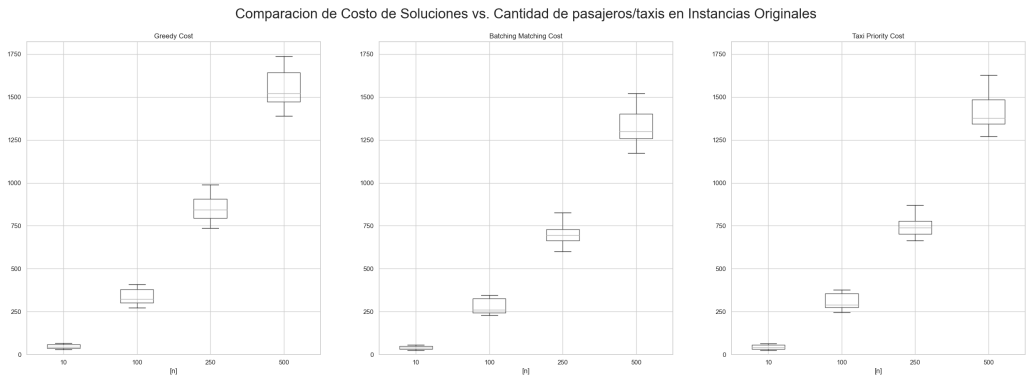
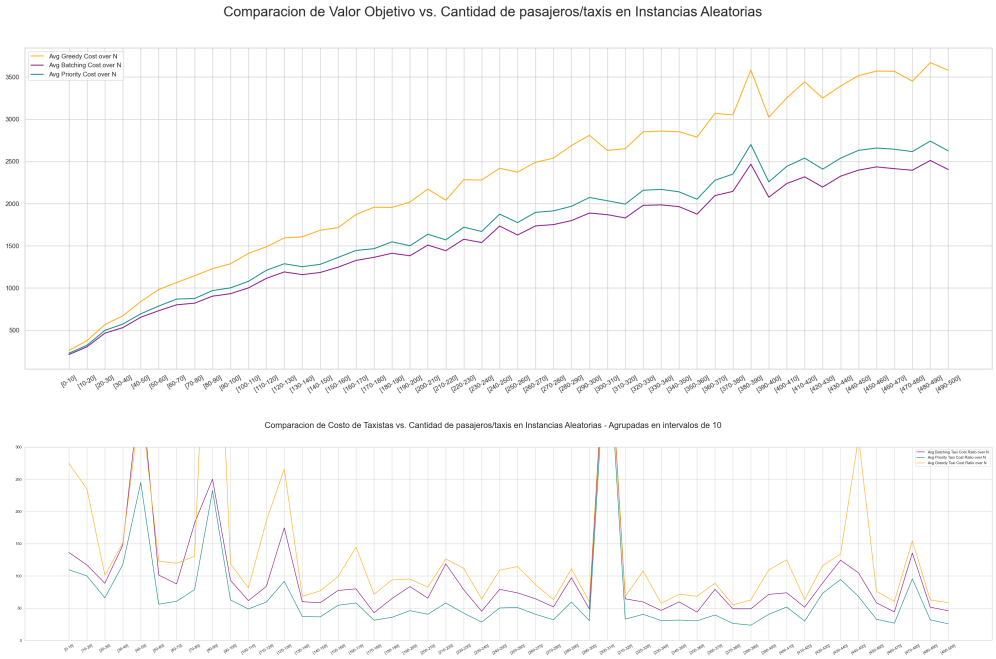


Figure 11: Desgloze Valor Objetivo - TaxiPriority vs Batching vs Greedy - Instancias Originales

Los resultados nos muestran como el modelo de `TaxiPriority` obtiene soluciones con un costo objetivo cercano, aunque mayor, al modelo de `Batching` pero de todas formas menor que el modelo `Greedy`. Como la resolución por `Batching` resulta en la asignación óptima, es esperado que su función de valor objetivo actúe de cota inferior para los otros modelos. Además, por la comparación hecha en la sección anterior entre la función de costo unitario de `TaxiPriority` y `Batching`, es esperado que el costo objetivo de `TaxiPriority` sea aunque mayor, similar al de `Batching`.

4.3.2 Comparación de soluciones en Instancias Aleatorias

Al igual que en la experimentación de `Batching`, se utilizaron las mismas ~ 5000 instancias generadas de forma aleatoria para comparar ahora con el nuevo modelo `TaxiPriority`.



Podemos observar el mismo comportamiento que aquel de las instancias originales, donde el modelo `TaxiPriority` obtiene soluciones con un costo objetivo cercano al modelo de `Batching` y menor que el modelo `Greedy`.

Como nuestro nuevo modelo resulta de una red de flujo máximo de costo mínimo de la misma cantidad de nodos y arcos que el modelo de Batching, sus complejidades algorítmicas son iguales.

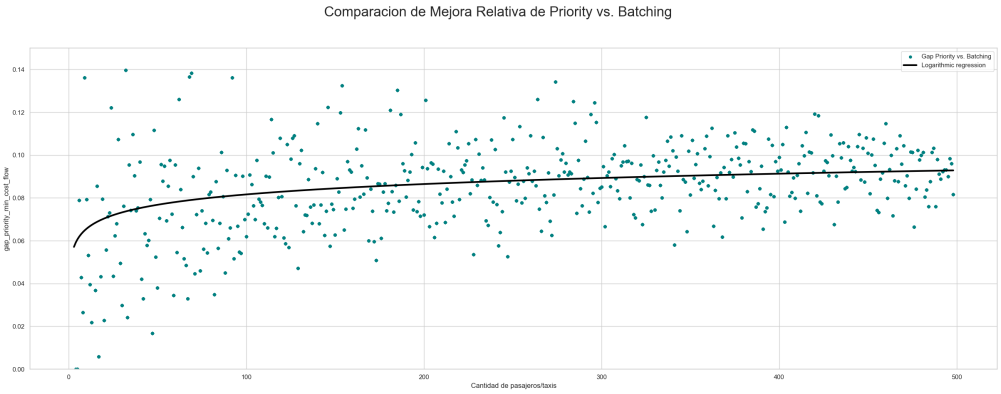


Figure 12: Mejora Relativa - TaxiPriority vs Batching - Instancias Aleatorias

Luego, observamos la mejora relativa que presenta `Batching` frente al nuevo modelo de `TaxiPriority`. La media de la mejora relativa entre `Batching` y `TaxiPriority` tiende a 0.092019, mientras que la mejora relativa promedio entre `Batching` y `Greedy` para la misma cantidad de taxis/pasajeros tiende a 0.5.

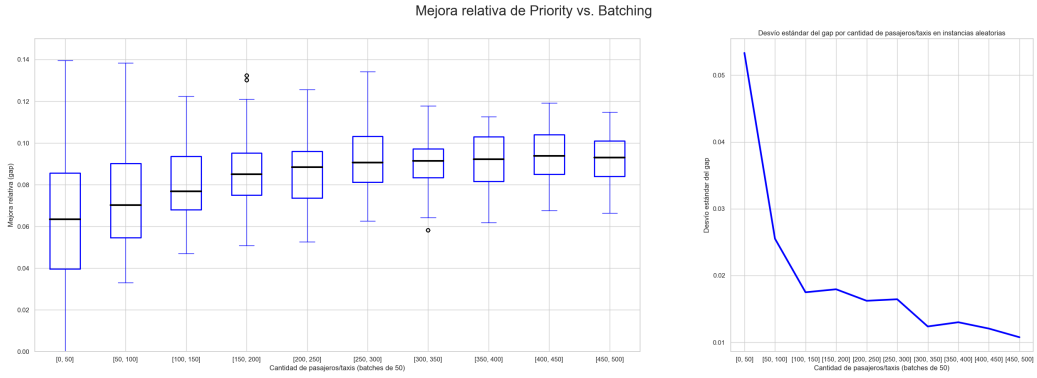


Figure 13: Mejora Relativa - TaxiPriority vs Batching std - Instancias Aleatorias

Ademas podemos ver como la desviación estándar de la mejora relativa entre `Batching` y `TaxiPriority` decrece a medida que aumenta la cantidad de taxis/pasajeros al igual que lo hace la desviación estándar de la mejora relativa entre `Batching` y `Greedy` (esto también se debe atribuir a la naturaleza aleatoria de las instancias).

4.3.3 Comparación de tiempos de ejecución entre `TaxiPriority` y `Batching Matching` en Instancias Aleatorias

En la experimentación se observan comportamientos análogos en cuanto a tiempos de ejecución entre `TaxiPriority` y `Batching Matching` ya que ambos modelos son eventualmente resueltos por el mismo algoritmo de resolución de flujo máximo de costo mínimo, para la misma cantidad de nodos y arcos, por lo que sus complejidades algorítmicas son iguales.

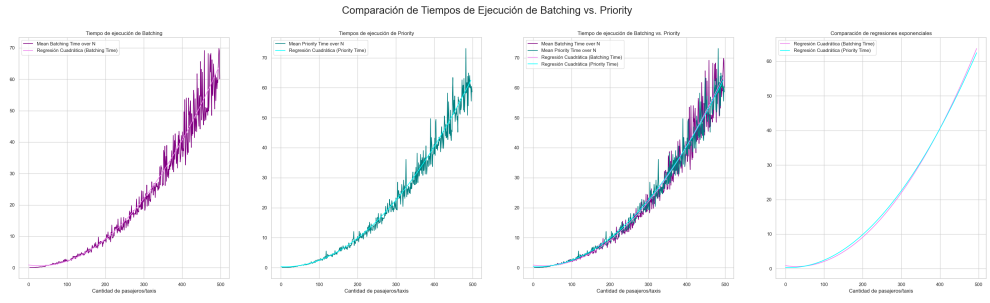


Figure 14: Tiempo de Ejecución - TaxiPriority vs Priority

5 Conclusiones

El objetivo de este trabajo fue el de modelar el problema de asignación de Taxis a Pasajeros como un problema de flujo de costo mínimo, y comparar los resultados obtenidos con los modelos de `Batching` y `Greedy` planteados. Ademas se propuso un nuevo modelo de asignación de Taxis a Pasajeros, llamado `TaxiPriority`, que prioriza los viajes más rentables para los conductores, y se compararon los resultados obtenidos con los otros dos modelos.

Una descripción mas informal para la forma en que los tres algoritmos distintos generan sus soluciones es la siguiente: El algoritmo FCFS (`Greedy`) prioriza los tiempos mas cortos para los pasajeros en orden de llegada, mientras que el modelo de `TaxiPriority` prioriza los viajes más rentables para los conductores. Por último, el modelo de `Batching` prioriza los viajes más rentables para la asignación total.

La curva de la mejora relativa del modelo de `TaxiPriority` tiene un comportamiento similar al del modelo de `Batching` en cuanto el Valor Objetivo resultante para cada instancia comparada contra el algoritmo `Greedy`. Con respecto a la relación entre el algoritmo `Greedy` y el `Batching`, por definición el algoritmo `Greedy` obtiene una solución subóptima, por lo tanto su Valor Objetivo es mayor que el del modelo de `Batching` que optimiza esta métrica. Por lo tanto, la cota inferior de estos Valores Objetivos esta siempre delineada por aquella resuelta por el modelo de `Batching`.

Los resultados obtenidos en la experimentación muestran que el modelo de `TaxiPriority` es de todas formas una buena alternativa para la asignación de Taxis a Pasajeros si se busca este beneficio para los conductores, dado a que obtiene soluciones con un Valor Objetivo cercano al modelo de `Batching`.

En términos de tiempos de ejecución, el modelo de `Greedy` supera ampliamente a los otros dos modelos, resultado tambien esperado dado a que su computo esta dado por una complejidad algoritmica de $O(n^2)$, mientras que los otros dos modelos tienen una complejidad algoritmica de $O(n^2 * m * log(n * C))$. Ademas el modelo de `TaxiPriority` tiene un tiempo de ejecución similar al modelo de `Batching`, dado a que ambos modelos resuelven un problema de flujo de costo mínimo de la misma cantidad de nodos y arcos. En la experimentación se observa como el modelo de `Greedy` resuelve las instancias en ordenes de magnitudes menores que los otros dos modelos.

Ambas experimentaciones se vieron beneficiadas por la generación de instancias aleatorias, dado a que se pudo observar que: el comportamiento de los modelos en un rango más amplio y continuo de instancias; los resultados obtenidos en las instancias originales se mantienen en las instancias aleatorias, los modelos de `Batching` presentan una mejora relativa proyectada con respecto al modelo de `Greedy` en cuanto a la función objetivo; y el modelo de `TaxiPriority` presenta un comportamiento similar al modelo de `Batching` en cuanto a la función objetivo.

6 Referencias

- [1] Uber Matching Batching https://www.uber.com/us/en/marketplace/matching/
- [2] Min Cost Flow - OR-TOOLS https://developers.google.com/optimization/reference/graph/min_cost_flow
- [3] Finding Minimum-Cost Circulations by Successive Approximation Andrew V. Goldberg, Robert E. Tarjan, (1990) Finding Minimum-Cost Circulations by Successive Approximation. Mathematics of Operations Research 15(3):430-466.