UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

PROGRAMA DE MAESTRÍA Y DOCTORADO EN CIENCIAS MATEMÁTICAS Y DE LA ESPECIALIZACIÓN EN ESTADÍSTICA APLICADA

SIMULACIONES EN REDES VIALES: IMPACTO DE LA ARQUITECTURA DE RED EN EL FLUJO VEHICULAR.

TESIS
QUE PARA OPTAR POR EL GRADO DE:
MAESTRA EN CIENCIAS

PRESENTA:
LUZ MARIANA BLAZ CARRILLO

DIRECTOR DR. RICARDO ATAHUALPA SOLÓRZANO KRAEMER
FACULTAD DE CIENCIAS

CIUDAD DE MÉXICO, FEBRERO DE 2024.

A mi familia y amigos amados.
A los profesores que han contribuido en mi formación desde el principio de mi vida.

Índice general

Introducción

Para las personas que conocemos la Ciudad de México (CDMX), y más aún para los que vivimos en ella, queda claro que el transporte en automóvil es un problema; ya que los trayectos de un punto a otro de esta capital toman demasiado tiempo (de acuerdo al *TomTomTraffic Index*, los tiempos de recorrido toman hasta 20 minutos extra por cada 10 km respecto a lo que tomarían en calles despejadas [33]); y hay largos periodos durante los viajes, en los que la velocidad en que se mueven los autos es extremadamente baja, llegando a los 13 km/h aproximadamente, según la misma fuente.

¿Por qué sucede esto? Hay una respuesta lógica: La CDMX está congestionada y eso está claro. Según el INEGI con datos del 2020, en la ciudad hay actualmente de 9.2 millones de habitantes [16] y 6.2 millones de autos en el parque vehicular registrado en el 2021 [17] lo que se traduce en 68 autos por cada centenar de habitantes, esto sin contar los autos no registrados en la Ciudad de México que circulan diariamente por ella.

La congestión del tráfico vehicular es problemática porque empeora la calidad de vida de los habitantes de la ciudad, a través de la pérdida de tiempo en los trayectos, la contaminación que genera y el incremento en el número de accidentes que se producen. Si sugerimos que la congestión vehicular es un problema, una posible solución sería una estrategia para reducir esta congestión, por ejemplo, limitando el número de vehículos en las calles o el máximo número de vehículos por familia. O aún yendo más allá, se podrían implementar políticas que reduzcan la población de la ciudad, descentralizando las fuentes de empleos y educación en el país. Sin embargo estas medidas son difíciles de implementar, es por ello que en este trabajo se explora una alternativa. ¿Qué pasa si no podemos reducir el número de automóviles pero sí podemos modificar algunas propiedades en el espacio a través del que circulan los autos?

En este trabajo de tesis se presenta un estudio de redes tipo Manhattan con algunas modificaciones inspiradas en arterias características de la Ciudad de México. El objetivo es analizar en qué condiciones de densidades de automóviles éstas arterias resultan en una ventaja para los conductores, especialmente analizando si hay una mejoría en su tiempo y velocidad de recorrido. En el análisis se incluyen desde densidades bajas de automóviles, en las que la mayoría de ellos puede circular en la velocidad máxima permitida, hasta densidades en las cuales llega un momento de saturación total en la

red, es decir, donde, debido a las constricciones físicas de las calles, ya no hay ningún movimiento posible para ningún automovilista.

El estudio se realizó a través de simulaciones computacionales, considerando que los conductores son agentes que interactúan en una red embebida en un espacio físico y cuya principal regla de decisión es la de actuar buscando la optimización de su tiempo idividual de recorrido, sin interés en una optimización de los tiempos colectivos. Las entidades individuales consideradas (conductores) tienen un origen y un destino propios en la red y sólo están presentes en el espacio físico de interacción, desde que deben partir de su origen, en un tiempo determinado, y hasta que logran llegar a su destino.

En este trabajo asumimos que el tráfico de automóviles es un sistema complejo y que, por tanto, la interacción de estos agentes se traduce en propiedades emergentes del problema a gran escala, es decir, en flujos y regiones de la red con alta densidad. Es por ello que aunque nuestras reglas son sobre el comportamiento individual de los conductores y las propiedades de las redes por las que viajan, nuestros análisis serán útiles para entender el efecto de la inclusión de arterias en una red de otro modo homogénea.

El modelo está implementado en el lenguaje de programación `Julia` y se utiliza en múltiples simulaciones de un mismo sistema evolucionando a través del tiempo. Estas simulaciones nos permiten tener una comprensión más exhaustiva del problema y nos orientan acerca de la relación entre las densidades de automóviles, la arquitectura de red y parámetros macroscópicos, como los tiempos de recorrido y las velocidades promedio. Nuestro modelo fue construido a partir de preguntarnos cuáles son las principales métricas de importancia para definir el estado de tráfico de una ciudad, qué tipo de reglas pueden implementarse a los automovilistas para lograr resultados que se asemejen a la realidad y cómo podemos implementar diferentes arquitecturas de red.

La estructura del trabajo incluye cuatro capítulos teóricos que definen, en ese orden: Conceptos de teoría de redes, fundamentos del estudio del tráfico, fundamentos de algoritmos, estructuras de datos y algoritmos en gráficas. Estos capítulos darán una base teórica para explicar las decisiones que tomamos en las simulaciones de los diferentes escenarios considerados. Posterior a esta sección, se especificarán los tipos de redes con arterias que estudiaremos y se darán detalles de su implementación computacional. Finalmente se presentará un análisis de resultados y se presentarán las conclusiones.

Analizando los resultados obtenidos haremos explícitos los puntos donde para cada arquitectura ocurre un cambio de fase en el estado de saturación del sistema y mostraremos cómo en esos puntos suceden comportamientos interesantes para todos los parámetros medidos (velocidades, distancias y tiempos de recorrido, entre otros). Con esta información concluiremos que la existencia de arterias en una red tipo Manhattan, para ciertas densidades de automóviles, pueden ser un problema más que una solución, debido a que el ofrecer una alternativa de mucho menor costo entre muchas otras de valor equivalente, la mayoría de las personas buscarán beneficiarse de ella, provocando que el sistema que incluye a esta alternativa tenga un produzca en las personas un rendimiento promedio aún peor que el que no la incluye.

Capítulo 1

Teoría de redes

El estudio del sistema de tráfico vehicular requiere de una abstracción del espacio físico donde estos conductores existen e interactúan. Este espacio físico será modelado a través de gráficas dirigidas con características propias (redes). Por esta razón es importante definir a las redes como un objeto matemático, así como establecer algunas de sus características.

1.1. Teoría de gráficas y definiciones importantes

Las redes pueden entenderse de diferentes maneras, pero en general pueden representarse como una colección de elementos y las conexiones entre ellos [14]. Por ello es que, en general, las redes son representadas como gráficas (grafos). En este trabajo en particular se entenderá como **red** a una gráfica dirigida que contiene cierta información sobre el sistema que se está modelando. Para poder entender esto, la forma en que esta información se almacena y algunas decisiones que se tomaron en la forma de realizar los algoritmos es importante tener bases sólidas en las definiciones primordiales de las redes y sus características como objetos matemáticos. Es por ello que comenzaremos por las definiciones más básicas.

Una gráfica se define como un par ordenado de dos conjuntos (V, A), V debe ser no vacío, y sus elementos serán llamados **vértices** o **nodos**, A no debe cumplir ninguna condición sobre su vacuidad, sus elementos serán llamados **aristas**, y consistirán en pares de vértices.

- Si los elementos de A son pares no ordenados, se dice que la gráfica es **no dirigida**. Mientras que si lo son, se considerará que la gráfica es **dirigida** ó **digráfica**.
- Si se cumple que $\forall a = (u, v) \in A$, $u \neq v$ y $\nexists a' \in A \mid a = a'$, entonces decimos que la gráfica es **simple**.

Las gráficas se pueden visualizar como puntos (correspondientes a los vértices) y líneas entre ellos, que representan las aristas, en el caso de gráficas no dirigidas. Para

las gráficas dirigidas se utilizan flechas en vez de líneas, para una arista que es de la forma (u, v) se dibujará una flecha que sale de u y termina (tiene su punta) en v. Puesto que en este proyecto se está realizando un modelo de las calles de la Ciudad de México, y las calles tienen asignadas direcciones sobre las que se puede transitar, las gráficas que utilizaremos aquí serán gráficas simples y dirigidas. A continuación daré algunas definiciones útiles para el trabajo, estas definiciones fueron extraidas de los libros: *Chromatic Graph Theory* de Chartrand y Zhang [8], de *Graph Theory* de Bondy y Murty [4] y de *Networks, An Introduction*, de Newman [22].

1.1.1. Orden y tamaño

El **orden** de una gráfica $G = (V, A)$ es $|V|$, es decir el número de vértices. Mientras que el **tamaño** de G es $|A|$, el número de aristas.

1.1.2. Relaciones entre vértices y aristas

Si $G = (V, A)$ es una gráfica no dirigida y $a = (u, v) \in A$ se dice que u y v son **puntas** de a, o que u y v **son incidentes** a a y viceversa. Si $D = (V, A)$ es una gráfica dirigida y $a = (u, v) \in A$, se dice que a **une a** u **con** v. También se dice que $u \in V$ es la **cola** de a y que $v \in V$ es su **cabeza**. Tanto u como v pueden ser llamados una **punta** de a.

1.1.3. Vecinos y vecindades

Si $a = (u, v)$ es una arista de G y G es simple y no dirigida, entonces u y v son **vecinos**, siendo la propiedad **ser vecino de**, una relación simétrica y reflexiva. El conjunto de vecinos de un vértice v es llamado **la vecindad de v**, y denotado como $N(v)$.

En el caso donde $D = (V, A)$ es una gráfica dirigida y $a = (u, v) \in A$, no se habla de vecinos y vecindades, dado que esta definición no contempla que el par (u, v) sea ordenado. Para este caso en especial diremos que u es un **invecino** de v y que a su vez v es un **exvecino** de u. El conjunto de invecinos de un vértice v es llamado, en este caso, **invecindad de v**, y denotado como $N_D^-(v)$, mientras que el conjunto de exvecinos es llamado **exvecindad de v**, y denotado como $N_D^+(v)$.

1.1.4. Grado, ingrado y exgrado

Si $G = (V, A)$ es una gráfica no dirigida, **el grado de un vértice** $v \in V$ es el número de vértices vecinos de v, es decir la cardinalidad de $N(v)$. De manera análoga, para una gráfica dirigida $D = (V, A)$, **el ingrado de** $v \in V$ es $|N_D^-(v)|$ y **el exgrado de** v: $|N_D^+(v)|$.

Cabe mencionar que para cualquier digráfica D, es posible asociar una gráfica G con el mismo conjunto de vértices V y sustituyendo cada par ordenado $a \in A$ por un par no ordenado. Dicha gráfica es llamada **gráfica subyacente** de D ($G(D)$). La posibilidad de obtener una gráfica a partir de una digráfica, permite utilizar para las digráficas algunos conceptos definidos para gráficas. Por ejemplo, si se habla del grado de un vértice v tal que $v \in V$ de una digráfica $D = (V, A)$, se entiende que se hace referencia al grado de dicho vértice en $G(D)$, lo mismo si se habla de vecindades y vecinos. Sin embargo no siempre es posible hacer lo opuesto, es decir tomar conceptos de digráficas y aplicarlos a una gráfica, ya que muchos de ellos dependen de una orientación de las aristas.

1.1.5. Caminos y ciclos

Hay diferentes formas de describir secuencias de vértices de una gráfica a través de las cuales uno se puede mover a través de una gráfica o digráfica. Primero se darán las descripciones de estas estructuras para una gráfica no dirigida y luego para una dirigida:

Sea $G = (V, A)$ una gráfica simple no dirigida y $u, v \in V$ no necesariamente distintos. Un **camino** de u a v es una secuencia W de elementos en V, que comienza en u y termina en v, que cumple la propiedad de que para cualesquiera elementos consecutivos en W, estos son vértices adyacentes de G. Un camino W se puede expresar como $W = (u = v_0, v_1, ..., v_k = v)$ donde $(v_i, v_{i+1}) \in A$, $0 \leq i \leq k-1$. El número total de aristas encontradas en W, incluyendo posibles aristas repetidas, se conoce como **la longitud** de W. Un camino en el cual el vértice inicial y el final son diferentes se conoce como **camino abierto**, en el caso contrario se habla de un **camino cerrado**.

Hay algunos nombres particulares para caminos abiertos:

- Un **paseo** es un camino en el cual no hay aristas repetidas, pero para el cual puede haber vértices repetidos, esto quiere decir que para $W = (u = v_0, v_1, ..., v_k = v)$ no habrá ninguna pareja de elementos sucesivos v_i, v_{i+1} para el cual suceda que existe otra pareja de elementos sucesivos v_j, v_{j+1} que también se encuentre en W.
- Una **trayectoria** es un camino en el cual no se repite ningún vértice.

Y algunos para caminos cerrados:

- Un **circuito** es un camino cerrado en el cual no hay aristas repetidas, pero para el cual puede haber vértices repetidos.
- Un **ciclo** es un camino cerrado en el cual no se repite ningún vértice, salvo el inicial (que es igual al final). Formalmente $W = (v = v_0, v_1, ..., v_k = v)$ donde $\nexists i, j | 0 \leq i \leq k-1, 0 \leq j \leq k-1, i \neq j$ tales que $v_i = v_j$.

Para el caso de una digráfica $D = (V, A)$, las definiciones son muy similares. En este caso:

- Un **camino dirigido** entre dos vértices $u, v \in V$ es una secuencia $W = (u = v_0, v_1, ..., v_k = v)$ donde (v_i, v_{i+1}) es una flecha en A.
- Un camino dirigido sin ninguna arista repetida es un **paseo dirigido**.
- Un camino dirigido en el que no se repite ningún vértice es **una trayectoria dirigida**.
- Un camino dirigido que comienza y termina en el mismo vértice es un **circuito dirigido**, y
- Un circuito dirigido que no repite ningún vértice más que el inicial y final es un **ciclo dirigido**.

Cuando se está hablando en general únicamente de gráficas dirigidas, es posible omitir la parte de "dirigido"de las definiciones anteriores y se entenderá que nos referimos a las definiciones para digráficas y no a las mismas para la gráfica subyacente.

1.1.6. Componentes y conexidad

Una **componente** de una gráfica no dirigida $G = (V, A)$ se define como un subconjunto W de V tal que entre todos los vértices $v \in W$ existe al menos una trayectoria, que además cumple la propiedad de que ningún otro vértice en $V \setminus W$ puede ser añadido conservando la propiedad de existencia de trayectoria. Una gráfica es **conexa** si tiene sólo una componente.

Para una digráfica $D = (V, A)$ las definiciones de componente y de conexidad son un poco más complicadas, puesto que para casos como el de una red de una ciudad, no sólo nos importa que la gráfica subyacente sea conexa sino que efectivamente uno pueda llegar de un vértice a otro *siguiendo la dirección de las flechas*. También hay que tener cuidado con la simetría ¿Un vértice u y otro v están conectados si sólo es posible ir de u a v y no al revés?

Para esto se dan ahora varias definiciones para una digráfica $D = (V, A)$:

- Dos vértices en V pertenecen a la misma **componente débilmente conexa** si pertenecen a la misma componente conexa de la gráfica subyacente de D. una digráfica es **débilmente conexa** si tiene sólo una componente débilmente conexa.
- Dos vértices $u, v \in V$ están fuertemente conectados si existen una uv trayectoria y una vu trayectoria. Una **componente fuertemente conexa** es un conjunto maximal de vértices fuertemente conectados entre sí. Una digráfica es fuertemente conexa si tiene sólo una componente fuertemente conexa.

Consideramos que para las simulaciones de tráfico en la ciudad toda esquina debe ser accesible desde cualquier otra, por eso las redes que se utilizarán como modelos de las calles y esquinas de una ciudad son redes dirigidas fuertemente conexas.

1.1.7. Distancia

La distancia entre dos nodos $u, v \in V$ de una gráfica $G = (V, A)$ (dirigida o no) es la longitud mínima entre las longitudes de las trayectorias entre u y v. La distancia de un vértice a sí mismo es 0 y la distancia de vértices que pertenecen a diferentes componentes conexas es ∞.

1.1.8. Árboles

Esta sección es principalmente para hablar de una estructura de datos homónima. Una gráfica no dirigida $G = (V, A)$ conexa (tal que existe un camino de u a v entre cualquier par de vértices en V) que no tiene ningún ciclo es conocida como **árbol**. De hecho los árboles pueden caracterizarse por dos de tres propiedades (es decir que siempre que una gráfica G cumple dos de ellas es un árbol):

1. G es conexa
2. G no tiene ciclos
3. Si m es el tamaño de la gráfica y n su orden: $m = n - 1$.

Para el tema de estructuras de datos del que se hablará en un capítulo subsecuente, introduciremos el concepto de **raíz** y **árbol con raíz** ó **enraizado**: La raíz es un nodo particular que se selecciona del árbol [7] y en general la representación de un árbol enraizado es con la raíz en cierta posición y sus nodos vecinos en una posición inferior inmediata, después, para cada nodo, se sigue representando a sus vecinos en una posición inferior inmediata. En general podemos definir la relación entre nodos adyacentes como de "padre e hijo", donde el padre es el nodo más cercano a la raíz y el hijo es el más alejado de la raíz.

Para las siguientes dos subsecciones, además de la consulta a [4], también se revisaron textos más enfocados a la teoría de redes [22], así como a algoritmos relacionados con gráficas [13].

1.1.9. Representación matricial de gráficas

Como se especifica en [4], las gráficas son usualmente representadas con dibujos, sin embargo esta representación no es útil para su manipulación computacional. Para este tipo de aplicaciones se suelen utilizar dos estructuras matriciales, que permiten representar unívocamente a una gráfica:

1. La **matriz de incidencia** de una gráfica $G = (V, E)$ es una matriz de $n \times m$ (donde n es el orden y m el tamaño de la gráfica) que representa el número de veces que un vértice es incidente a una arista. Específicamente, la matriz de incidencia es $M_G = (m_{ve})$ con m_{ve} representando la cuenta de cuántas veces el

nodo v incide en la arista e. En el caso de que en G exista una arista de un vértice v a sí mismo (estructura llamada **lazo**), la incidencia se cuenta doble. Sin embargo este tipo de estructura no se encuentra presente en las gráficas que utilizaremos en este trabajo.

2. La **matriz de adyacencia** de una gráfica $G = (V, E)$ es una matriz de $n \times n$ donde n es el número de vértices en V. Esta matriz representa cuántas aristas unen a diferentes parejas de nodos. Específicamente la matriz de adyacencia es $A_G = (a_{uv})$ con a_{uv} el número de aristas que unen a u con v. Nótese que, para gráficas simples, a_{uv} siempre tomará los valores de cero o uno.

Estas definiciones son para una gráfica no dirigida, pero pueden ser extendidas a una gráfica dirigida, definiendo la matriz de incidencia como el número de veces que un vértice es cabeza (o cola) de una flecha. Para la matriz de adyacencia la información no cambia, simplemente en este caso la matriz no tiene que ser simétrica (por no serlo la relación de incidencia).

Cabe resaltar que en general la matriz de adyacencia es más pequeña de la de incidencia y por tanto es más usual usar matrices de adyacencia como representación de gráficas. Este tipo de matrices son precisamente las que se usarán en este trabajo.

1.1.10. Gráficas con pesos en las aristas

Es posible asignar pesos, o valores numéricos a las aristas, para representar propiedades tales como la distancia, el costo de pasar por una arista, etc. La inclusión de esta información puede realizarse a través de utilizar una extensión de la matriz de adyacencia, substituyendo a_{uv} por el peso de la arista w_{uv}. En esta representación, para dar a entender que una arista no existe entre u y v, se coloca un valor igual a infinito [13].

En general una gráfica con pesos en las aristas se entiende como una gráfica con una función de pesos: $G = (V, A, w)$ donde V son los vértices, A las aristas y w es una función de las aristas en los reales. Esto es válido para gráficas dirigidas o no dirigidas. En este trabajo hablaremos indiferentemente de gráficas con pesos como $G = (V, A, w)$ o como su representación matricial.

1.2. Flujos en redes

Esta sección se basa en una revisión de [2]. La idea es ver cómo podría haber transporte (o flujo) en las flechas de una digráfica, que en este contexto llamaremos red. Se verá que es posible encontrar cuál es el flujo máximo que puede haber entre dos vértices, sabiendo que cada arista tiene cierta capacidad de carga (que soporta hasta cierta cantidad de objetos "pasando" a través de ella) o bien cuál es el flujo de mínimo costo, considerando que pasar por una arista tiene un precio asociado.

1.2.1. Problema del flujo con costo mínimo

El problema puede resumirse como: *Si el flujo a través de las flechas de una red tiene un costo unitario asociado y debemos enviar flujos a partir de un vértice (o varios) y hacia otro (o varios), ¿Cómo podemos satisfacer las condiciones de envío y recepción al mínimo costo posible?*

La formulación sería la siguiente: Sea $G = (V, A)$ una gráfica dirigida, con un costo asociado a cada flecha c_{ij} y una capacidad de cada flecha u_{ij}. También es posible incluir una cota mínima por flecha l_{ij} (la cantidad de flujo que tiene que haber). Para cada nodo $i \in V$, se asocia un número entero $b(i)$ que representa su demanda o producción:

- Si $b(i) < 0$ Se trata de un nodo con demanda o **sumidero**.
- Si $b(i) > 0$ Se trata de un nodo con producción o una **fuente**.
- Si $b(i) = 0$ es un **nodo intermedio**.

La variable a optimizar es el flujo que pasa por cada flecha x_{ij} y la cantidad a minimizar:

$$\sum_{(i,j)\in A} c_{ij}x_{ij}$$

Con las constricciones:

$$\sum_{j|(i,j)\in A} x_{ij} - \sum_{j|(j,k)\in A} x_{jk} = b(j); \forall i \in V$$

$$l_{ij} \leq x_{ij} \leq u_{ij}; \forall (i,j) \in A$$

A la primera de esas constricciones se le conoce como **constricción de conservación de masa**, a la segunda como **constricción de cotas de flujo**.

Es importante resaltar que la suma de los $b(i)$ a través de todos los nodos de la red debe ser 0, es decir que no hay flujo que pueda estancarse en la red, ni puede ocurrir que no haya flujo suficiente para la demanda total.

1.2.2. Problema del flujo máximo

El problema puede resumirse como: *Si una red tiene una capacidad máxima en sus aristas, ¿Cómo podemos transportar el mayor flujo posible entre un par de vértices, respetando las capacidades?.*

En esencia tomamos dos nodos s y t de una digráfica, de tal forma que s es una fuente y t un sumidero y queremos enviar la máxima cantidad de flujo (estacionario) entre s y t, con una restricción de capacidad máxima de las aristas u_{ij}.

Este problema puede formularse como un caso particular del problema de costo mínimo con los siguientes valores:

- $b(i) = 0$ para todos los nodos.
- $c_{ij} = 0$ para todas las aristas.

Después se introduce una arista adicional de t a s con costo $c_{ts} = -1$ y capacidad $u_{ts} = \infty$. Con estos cambios, la solución del problema del flujo de costo mínimo maximiza el flujo en la arista (t, s), pero, como en esta representación s y t son nodos intermedios, todo el flujo a través de ambos debe compensarse, por lo que en realidad estaremos obteniendo el máximo flujo posible de s a t en la red original.

1.2.3. Limitaciones de los algoritmos de flujo máximo y de flujo con costo mínimo

Los algoritmos de flujo máximo y de flujo de mínimo costo tienen una amplia variedad de aplicaciones, por ejemplo, los problemas en los cuales hay que asegurar una entrega óptima de bienes o servicios a localidades específicas, el flujo óptimo de materiales a través de máquinas, el ruteo de llamadas a través de un sistema telefónico e, inclusive, el flujo de automóviles en un estado estacionario.

La condición del estado estacionario es una de las principales limitaciones de este tipo de algoritmos para un modelo como el que se presenta en este trabajo porque, en esencia, el tráfico en una red de automóviles que representa una ciudad no suele llegar a un estado estacionario, los autos parten de un origen y llegan a un destino propio, no toda la gente sale del mismo lugar ni llega al mismo lugar y en general la ciudad se encuentra en estados distintos durante el transcurso de un día.

Otras posibles limitaciones relacionadas con este proyecto son:

- Que en general el costo de las aristas aumenta conforme aumenta el flujo a través de ellas de una forma no lineal, esto podría ser resuelto con una extensión de los problemas aquí descritos: Resolviendo el **problema del flujo de costo convexo**, sin embargo aún éste presenta la limitante de representar soluciones en un estado estacionario.
- Que en general en este tipo de problemas y los algoritmos para resolverlos se espera que haya pocos nodos fuente y sumidero y en los problemas que nos interesan casi todos los nodos de hecho son, en algún momento fuentes o sumideros de autos.
- Que para este tipo de problemas se espera que el flujo se encuentre en movimiento todo el tiempo, es decir que no considera que en una arista intermedia del trayecto de un automovilista, este pueda esperar a que se descongestione el trayecto y después seguir avanzando.

En la sección 4.2 del capítulo de algoritmos se explica muy brevemente la idea de cómo se resuelve este tipo de problemas, sin embargo no se profundiza tanto en

ello, puesto que para nuestra aplicación en particular estos algoritmos se encuentran severamente limitados para simular los casos que son de interés.

Con esto terminamos las definiciones y conceptos de teoría de redes que serán utilizadas en el trabajo de tesis, a continuación, se presenta un capítulo sobre los fundamentos del estudio del tráfico.

Capítulo 2

Fundamentos del estudio del tráfico vehicular

En este capítulo revisaremos los enfoques con los que se modela el tráfico vehicular y explicaremos por qué se considera un sistema complejo. Posteriormente revisaremos conceptos relacionados a mediciones que pueden hacerse sobre un sistema de tráfico. Finalmente hablaremos sobre las consecuencias que puede tener la forma de toma de decisiones egoísta sobre un sistema de tráfico vehicular.

2.1. Sistemas complejos

Los sistemas complejos pueden definirse como un conjunto de componentes interactuantes, cuya interacción es típicamente no lineal. Este tipo de sistemas pueden surgir y evolucionar a partir de la autoorganización, de forma tal que una red que pueda representar al sistema no es completamente aleatoria, ni completamente regular. Esta estructura permite el desarrollo de comportamientos emergentes en una escala macroscópica [28]. Las características fundamentales de un sistema complejo son:

- La capacidad de autoorganización
- La aparición de comportamientos emergentes a macro escala
- La interacción no lineal y sinergia entre las componentes

Bajo esta definición, los flujos de automóviles en las ciudades pueden considerarse un sistema complejo, puesto que están involucrados múltiples factores, incluyendo vehículos, infraestructura, comportamientos de los conductores y condiciones ambientales que interactúan en una manera no lineal y sinérgica.

En un sistema de tráfico vehicular, un cambio pequeño en algún parámetro puede tener efectos significativos en el comportamiento general, siendo un cambio pequeño una

colisión, el cierre de una vía o la falla técnica de algún semáforo. Además de esto, las decisiones individuales de los conductores (desde las rutas elegidas hasta los momentos donde frenan y aceleran) se propagan a través del sistema y afectan el flujo para todos los demás.

2.2. Revisión de modelos de tráfico vehicular preexistentes

Esta sección se redactó principalmente partir de una revisión de [36] y de la compilación de *Equilibrium and Learning in Traffic Networks* [9].

2.2.1. Tráfico vehicular

El tráfico vehicular puede ser descrito como un estado estacionario que surge del comportamiento adaptativo de conductores egoístas que desean minimizar sus tiempos de viaje mientras compiten por una capacidad de calles limitada [9]. En esta visión, los conductores eligen los caminos de menor tiempo y aprenden de las condiciones del flujo. A su vez, la congestión se modela como tiempos de viaje que incrementan conforme aumenta el flujo que lleva cada arista. También es posible incluir asunciones de heterogeneidad en la toma de decisiones de los conductores, fluctuaciones aleatorias que indiquen cómo funciona la toma de decisiones entre rutas de costos virtualmente iguales y percepciones individuales de la red y las rutas basadas en las experiencias pasadas de cada conductor.

2.2.2. Modelos de tráfico vehicular

En [36], se da una amplia revisión de los enfoques con los que normalmente se modela el tráfico vehicular. En esencia todas las familias de modelos de tráfico se derivan del trabajo original de Greenshields en 1934, conocido como el diagrama fundamental, en el cual se relacionaba el espaciamiento entre dos vehículos y su velocidad. Después de esta introducción, diferentes diagramas se han introducido, tratando de relacionar diferentes variables del tráfico, incluyendo velocidades, densidades, flujos y tiempos. Además de que diversas formas funcionales han sido propuestas para el desarrollo.

Además de los estudios de tipo Diagrama Fundamental (o FD por sus siglas en inglés), existen otras tres familias que agrupan los modelos de tráfico, y que se diferencian por la escala de las componentes con las que modelan: Modelos de flujo microscópico, modelos de flujo mesoscópico y modelos de flujo macroscópico.

Los modelos microscópicos describen el comportamiento de vehículos o conductores como individuos, en esta familia se incluyen los modelos con autómatas celulares (véase por ejemplo [19]), los modelos de distancia segura [21], de estímulo respuesta [34] y los modelos de punto de acción, que se asemejan a los anteriores pero consideran que

los conductores sólo pueden reaccionar a partir del punto donde un estímulo es lo suficientemente grande como para que puedan percibirlo.

Los modelos mesoscópicos describen el comportamiento de los vehículos en una forma agregada, pero el comportamiento individual de cada conductor está dado por reglas propias, los modelos mesoscópicos más conocidos son los de tipo cinética de gases, donde la distribución de la densidad y la de las velocidades, pueden ser utilizadas para describir las densidades y velocidades esperadas para el sistema completo.

Los modelos macroscópicos describen el tráfico como un flujo continuo y sólo se consideran variables agregadas (tales como velocidad promedio, densidad promedio y flujos).

2.3. Formas de medir el flujo vehicular en una ciudad

Para el estudio realizado, en el cual nos interesa medir parámetros macroscópicos del sistema, es posible determinar el estado del flujo de un sistema vehicular a través de métricas de los viajes de los conductores, es decir, distancias, tiempos y velocidades de recorrido promedio; a través de parámetros de red como congestiones de aristas ó señalando qué tan alejados se encuentran los parámetros medidos para un sistema en comparación con los parámetros del sistema optimizado, tal y como sucede con la métrica del Precio de la Anarquía que se discutirá en una sección posterior.

2.4. La paradoja de Braess

Esta sección está basada en el libro *Selfish Routing and the Price of Anarchy* de Tim Roughgarden [23], como indica el título del libro, la paradoja de Braess está relacionada con la toma de decisiones individual y egoísta.

Una decisión egoísta significa que fue optimizada para beneficio de quien la tomó, en el caso de elección de rutas. Una decisión egoísta es aquella que minimiza la distancia o el tiempo de recorrido del conductor. En un sistema donde cada conductor elige su ruta de esta forma es probable que si hay una ruta particularmente buena, en la cual se minimiza el tiempo o la distancia (como caso ejemplar: lo que sucede con las diagonales en redes que siguen una estructura Manhattan) la mayoría de los conductores para los que tenga sentido usarla por acortar su ruta, decidan hacerlo.

Sin embargo, cuando muchos conductores eligen la misma opción, es posible que esta se congestione, particularmente a densidades altas. ¿Qué pasará con los conductores entonces, algunos decidirán tomar rutas peores para el beneficio de la mayoría? Posiblemente no, en cuyo caso puede ser que inclusive el tráfico empeore a causa de la existencia de una ruta de mejor costo por sobre las demás. De esto habla precisamente la **paradoja de Braess**.

La paradoja de Braess fue descubierta en 1968 y se puede resumir como *"Cuando la toma de decisiones de rutas es egoísta, una mejora en la red puede degradar el*

desempeño de la red y a su vez el cierre de algunas calles puede mejorarlo" (a partir de [23] y [38]).

Un ejemplo simple de esta paradoja es el siguiente (a partir de [23]): Si existe un suburbio s y una estación de tren t y cierto número de conductores que desean llegar de s a t. Hay dos rutas que no interfieren entre sí, cada una compuesta de dos aristas, una de costo -o tiempo- constante (independientemente del flujo) y una cuyo costo aumenta linealmente con el flujo. El costo total de cada una de las opciones es $1 + x$, donde x es el flujo en esa ruta, como se observa en la imagen 2.1. Debido a que las dos opciones son idénticas, el tráfico se divide entre ellas equitativamente. Digamos que el tiempo promedio de s a t es entonces 90 minutos, que es el tiempo de recorrido por cualquiera de las dos rutas. Si se añade una arista que tiene un costo constante prácticamente cero entre los nodos intermedios, como en la imagen 2.2, la nueva ruta (s, v, w, t) es mejor que cualquiera de las anteriores, para flujos bajos. Si se está en una dinámica egoísta, todos los conductores ahora optarán por esta nueva ruta, el problema es que ahora el tiempo de llegada será mucho peor, ya que esta ruta se congestionará. De esta forma, añadir una nueva arista de costo 0 puede impactar negativamente todo el tráfico en la red.

Otro ejemplo es precisamente el trabajo de Youn, et al. [38] en el que se estudian rutas en calles principales de Londres y Nueva York, donde se encuentran calles que al ser removidas mejoran el flujo de automóviles de un punto a otro de la ciudad.

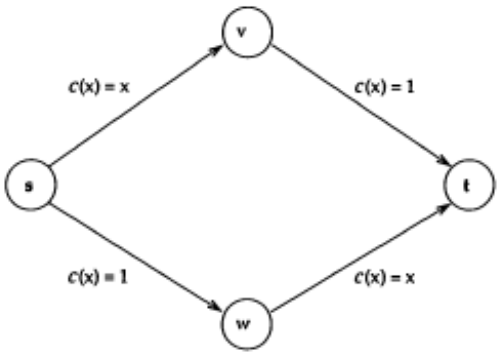

Figura 2.1: Estado inicial en la red en el ejemplo de la paradoja de Braess, extraído de [23].

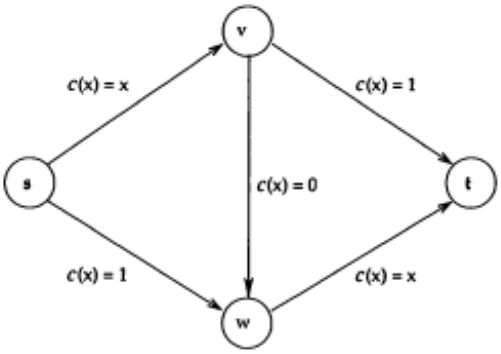

Figura 2.2: Estado de la red en el ejemplo de la paradoja de Braess después de agregar una nueva arista, extraído de [23].

2.5. El precio de la anarquía

El precio de la anarquía (o PoA, por sus siglas en inglés) es una medida de la ineficiencia del tráfico a causa de un planeamiento de rutas individual y egoísta. El PoA está formalmente definido como la razón entre el costo total de un sistema de tráfico bajo una optimización individual y el costo total del sistema en flujos que optimizan el costo, o bien como la razón entre el costo con un flujo en el equilibrio de Nash y el costo en un flujo óptimo (véanse [39], [23] y [37]). Esencialmente el PoA siempre toma un valor nunca menor a 1, pues lo mejor que puede llegar a ser un sistema es precisamente en el punto donde se optimizan los flujos para minimizar el costo, y puede llegar a tomar valores arbitrariamente grandes [26], conforme el ruteo egoísta se aleja del óptimo social. En el caso del trabajo el costo asociado a un estado en equilibro de flujos en la red es el tiempo promedio que le toma a los conductores recorrer su trayecto de origen a fin.

2.6. La ecuación del *Bureau of Public Roads*

Intuitivamente, es sabido que conforme una calle (arista) va alcanzando su capacidad máxima, la velocidad promedio en la misma decrece y por ello el tiempo para recorrerla aumenta. El *Bureau of Public Roads* (Buró de Caminos Públicos) en Estados Unidos, propuso una función de precisamente cuál es el aumento del tiempo de recorrido en una arista debido al flujo.

La función propuesta por el BPR es la siguiente:

$$l_{i,j} = d_{i,j}/v_{i,j}\left[1 + \alpha\left(\frac{f_{i,j}}{p_{i,j}}\right)^{\beta}\right]$$

donde $d_{i,j}$ es la distancia de la arista ij, $v_{i,j}$ es la velocidad máxima permitida, $f_{i,j}$ el flujo y $p_{i,j}$ la capacidad sobre la misma arista. Mientras que α y β son constantes positivas que se han determinado empíricamente en los valores $\alpha = 0.2$ y $\beta = 10$ para flujos congestionados [31].

Es el término $\left[1 + \alpha \left(\frac{f_{i,j}}{p_{i,j}}\right)^{\beta}\right]$ el que dicta cómo crece el tiempo conforme aumenta el flujo: Cuando el flujo es cero o el cociente entre el flujo y la capacidad $\left(\frac{f_{i,j}}{p_{i,j}}\right)$ es cercano a cero, el tiempo es el dado por la velocidad máxima permitida, que es lo que se espera en bajos flujos, donde uno puede circular libremente. Por otro lado, conforme satura una arista, el tiempo incrementa rápidamente.

Una vez que hemos explorado conceptos de tráfico, procederemos a hablar sobre algoritmos y estructuras de datos, pues es a través de ellos que podremos realizar las simulaciones de nuestro estudio.

Capítulo 3

Fundamentos de algoritmos y estructuras de datos

El objetivo de este capítulo es explicar la importancia del estudio de algoritmos y estructuras de datos, así como dar algunas definiciones importantes, con el fin de justificar los procedimientos computacionales utilizados en la modelación realizada en este trabajo. Muchas de las ideas vertidas en esta introducción están basadas en [10].

Informalmente un algoritmo es un proceso bien definido (inambiguo y preciso) que recibe cierto valor o conjunto de valores y a partir de ellos produce un resultado. Al conjunto de valores recibido se le llama *input* y al resultado obtenido se le llama *output*. El algoritmo es la secuencia de pasos que transforman un input en un output.

Un algoritmo también puede verse como una solución particular de un problema computacional, que es expresable, en la mayoría de las ocasiones, en términos de qué output se desea para ciertos inputs. Empleando esta última definición, podemos distinguir entre algoritmos correctos e incorrectos:

1. Los algoritmos correctos son aquellos que, dada cualquier *instancia del problema* (un input en concreto), proveen el output esperado.

2. Los algoritmos incorrectos son aquellos que para al menos una instancia del problema, devuelven un output que no es el especificado en el problema computacional.

Aunque pueda sonar extraño, un algoritmo incorrecto en muchos casos puede resultar funcional si es que logramos acotar su error. Exploraremos un poco este tema en la sección **algoritmos**.

Si analizamos la primera definición del algoritmo como una serie de pasos que se deben seguir para llegar del input al output, nos podemos dar cuenta de que hay muchas representaciones posibles para un algoritmo, tales como el lenguaje natural, que es en el que nos comunicamos los seres humanos (inglés, español, etc.), un código programado en un lenguaje de programación o inclusive el diseño de un circuito [12].

La lista de posibles problemas computacionales que se nos pueden ocurrir a los seres humanos es enorme, y más aún la lista de algoritmos que son una posible solución a dichos problemas, puesto que hay diferentes maneras de transformar el input en el output deseado. Un problema sencillo, como ordenar una secuencia finita de números del menor al mayor, tiene múltiples soluciones desarrolladas.

Entonces ¿Cómo decidimos qué solución es la que debemos implementar? La respuesta es que utilizaremos la solución más *eficiente* posible. La medida usual para la eficiencia de un algoritmo es la velocidad, es decir, cuánto tiempo se tarda la computadora en realizar la serie de pasos para llegar del input al output, y sobre todo nos interesa en qué medida crece el tiempo requerido para el proceso conforme crece el tamaño del input, de este tema hablaremos en la sección **complejidad computacional**. Siempre que sea posible escogeremos el algoritmo que tome menos tiempo, según la escala de las instancias del problema con las que estemos trabajando.

En la sección de **algoritmos**, se hablará de las técnicas más usuales para diseñar soluciones a problemas computacionales y se explicará cómo es posible evaluar las soluciones propuestas, para verificar que sean correctas y eficientes.

En la sección de **estructuras de datos** exploraremos distintas maneras en que pueden almacenarse y organizarse datos en la memoria de una computadora, con el fin de encontrar estructuras que nos permitan acceder y modificar los datos con facilidad, para utilizarlos eficientemente en los algoritmos que se implementen. Para problemas distintos, distintas estructuras de datos pueden resultar más eficientes que otras y ninguna estructura de datos es ideal para todos los problemas. Es por ello que es de suma importancia conocer las diversas estructuras de datos que han sido diseñadas, sus ventajas y limitaciones.

3.1. Complejidad computacional

Si las computadoras fueran infinitamente rápidas y la memoria infinitamente grande, cualquier algoritmo correcto sería útil para resolver un problema computacional. Sin embargo realizar las operaciones toma tiempo, el relacionado con los procesos físicos que deben ocurrir en los circuitos para transportar la información. Y la cantidad de información que puede guardarse también está limitada por los recursos que tiene una computadora.

Debido a esto es muy importante evaluar los recursos que un algoritmo requerirá, en general el principal recurso que evaluaremos es el tiempo que le toma a un algoritmo devolver un output dado un input. Si quisiéramos estimar el tiempo real, deberíamos conocer especificaciones de la computadora en la cual dicho algoritmo está corriendo, por ejemplo, la arquitectura del sistema, el compilador que estamos utilizando y detalles de la estructura física de la memoria del equipo. En la mayoría de los casos ni siquiera tenemos la información necesaria sobre estas especificaciones, además de que la estimación que hiciéramos sería válida sólo para un equipo en específico.

La manera en que podemos evitarnos esta problemática es pensando en una for-

ma con la cuál podamos dar estimaciones sin conocer la estructura específica de una computadora, y además podamos estimar el orden del tiempo que le tomará al algoritmo devolver soluciones a inputs de cualquier tamaño.

Para diferentes problemas *tamaño del input*, puede significar diferentes cosas: Si el input es un conjunto de números enteros, por el tamaño nos podemos referir a la cardinalidad de dicho conjunto, si el input es un par de enteros que deben multiplicarse, una buena medida de su tamaño es el número de bits que ocupa cada uno de ellos en la memoria. En el caso de algoritmos para los cuales el input es una gráfica, el tamaño puede ser el número de nodos y de aristas de la gráfica con la que estemos trabajando. Siempre que evaluemos el tiempo de cómputo dado el tamaño del input, deberemos especificar qué significa este tamaño.

La forma en la que lograremos calcular tiempos de cómputo sin meternos a los detalles de la computadora, es aprovechando la idea de que hay operaciones computacionales sencillas que toman tiempos constantes (o aproximadamente constantes para una misma computadora). Utilizando estas operaciones de tiempo constante como la base de algoritmos complejos, podremos calcular tiempos de cómputo contando cuántas operaciones de tiempo constante se tienen que realizar y obtendremos funciones que relacionen el número de pasos constantes que se tienen que hacer respecto al tamaño del input. Además no nos importará demasiado saber *cuánto tiempo* toma un algoritmo en regresar un output, sino *cómo crece el tiempo* respecto al tamaño del input.

Con esto aterrizamos en la forma específica en que vamos a describir los tiempos de cómputo de un algoritmo, calculando *tiempos de cómputo asintóticos*, lo que significa que describiremos el comportamiento de una función de crecimiento del tiempo a través de cotas asintóticas, utilizando las siguientes definiciones [24]. Diremos que:

- $f(n) = O(g(n))$ (que "f es O mayúscula de g") si existen constantes N y c, tales que $\forall n \geq N$, $f(n) \leq cg(n)$ en este caso, a partir de cierta N, $f(n)$ está acotada por arriba por algún múltiplo constante de $g(n)$. En este caso también se dice que "g domina a f".
- $f(n) = \Omega(g(n))$, si existen constantes c y N, tales que si $n \geq N$, $f(n) \geq cg(n)$, en este caso también se dice que "f crece al menos tan rápido como g"
- $f(n) = \Theta(g(n))$ si $f = O(g(n))$ y $f = \Omega(g(n))$, esto quiere decir que ambas funciones crecen a la misma tasa.
- $f(n) = o(g(n))$ ("f es o minúscula de g") si $\frac{f(n)}{g(n)} \to 0$ conforme $n \to \infty$, en este caso también se dice que "f crece estrictamente más lento que g".

Estos conceptos quedarán mucho más claros con el siguiente ejemplo. Un algoritmo que resuelve el problema computacional de obtener una lista con los primeros n números de la secuencia de Fibonacci. Esto significa que dado un número natural n (el input), el algoritmo debe entregar una lista que contenga los primeros n números de la secuencia de Fibonacci (el output). Para la *instancia del problema* $n = 5$, el output correcto deberá ser la lista: [0,1,1,2,3]. Hay muchas maneras de lograr esto, por

ahora no profundizaremos en la forma en la que podemos diseñar un algoritmo sino que analizaremos uno que corresponde a la forma "natural" de resolver el problema, el algoritmo puede ser descrito en *pseudocódigo* o en lenguaje natural, como se muestra a continuación:

Algoritmo 1: Algoritmo de Fibonacci

Datos: n la longitud del arreglo.
Resultado: Lista de los primeros n números de la secuencia de Fibonacci.
Fibonacci(n):
$F \leftarrow$ arreglo de ceros de longitud n
$F[0] \leftarrow 0$
$F[1] \leftarrow 1$
para *i=2* **a** n **hacer**
  $F[i] \leftarrow F[i-1] + F[i-2]$
fin

Ahora analicemos el tiempo de cómputo de este algoritmo, este análisis es independiente del equipo donde estemos llevando a cabo el proceso, o inclusive si lo estamos llevando a cabo a mano. La idea es contar cuántos pasos de tiempo constante tenemos que realizar para llegar del input al output. El proceso de asignar un valor a algún elemento de una lista toma un tiempo casi constante, al igual que realizar sumas y multiplicaciones de números de pocos dígitos, localizar un elemento con un índice específico en una lista (al menos para el caso del tipo de estructura de datos que se utiliza en este ejemplo llamada "arreglo"), entre otros.

Con esta información podemos ver que este proceso requiere:

- Crear un arreglo y asignar un cero a n espacios en la memoria, lo que toma n pasos constantes.
- Sustituir en la memoria dos elementos, lo que toma dos pasos constantes.
- Hacer avanzar a i $n-2$ veces
- Cada vez que avanzamos a i, en uno, sumar dos números, cuyo número de dígitos es proporcional a n[1], lo que toma n pasos. y asignar el resultado al i-ésimo elemento de la lista. Esta asignación toma un tiempo constante cada vez que movemos i, por lo que no aportará tiempo extra a la cuenta que estamos realizando.
- Devolver el resultado, lo que toma un paso constante.

Por lo tanto el número de operaciones de tiempo constante que se realizan es, en función de n: $f(n) = n + 2 + (n-2)n + 1 = n^2 - n + 3$, pero ¿para qué preocuparnos por las constantes? ya que no sabemos cuánto tiempo constante toma cada operación,

[1]Existe una relación entre el número de dígitos de un elemento en la serie de Fibonacci d y su índice en la serie n, donde $d(n) \approx n/5$, que puede obtenerse a partir de la fórmula de Binet [35] para números de Fibonacci y el teorema que establece que el número de dígitos de un entero n es $\lfloor \log_{10} n \rfloor + 1$.

es más fácil (y útil) notar que f es $O(n^2)$, puesto que para $c = 2$ y $n \geq 2$ sucede que $n^2 - n + 3 < cn^2$. También podemos asegurar que el algoritmo es $O(n^3)$, $O(n^4)$, etcétera, de hecho usando los mismos valores para c y N. En general esto no nos importa, para describir el tiempo de cómputo de un algoritmo, siempre trataremos de dar la cota más justa que podamos.

Habiendo dado este ejemplo de cómo utilizar la notación asintótica para el tiempo de cómputo, también llamada *complejidad computacional* de un algoritmo, queda más claro que utilizar este tipo de notación nos permite ahorrarnos muchos detalles y específicos del equipo con el cual estamos trabajando, para, en cambio, dar una noción general de cómo crece el tiempo que toma un algoritmo en llevarse a cabo. Como se establece en [11], cuando nos encontramos con una función de varios términos, con varias constantes multiplicándose, como $3n^2 + 4n + 5$, buscaremos reemplazar a la expresión con una cota asintótica justa, para este caso decimos que esta función es $O(n^2)$, porque la porción cuadrática de la suma domina los demás términos. A continuación se dan una serie de reglas comunes para utilizar este tipo de notación [11]:

- Las constantes multiplicativas pueden ser omitidas, podemos decir que $5n^3$ es $O(n^3)$.
- n^a domina a n^b, si $a > b$
- Cualquier término exponencial domina a cualquier término polinomial (3^n domina a n^a $\forall a$)
- Cualquier término polinomial domina a cualquier potencia de un logaritmo, n domina a $(\log n)^3$, y de la misma manera n^2 domina a $n\log n$

La notación asintótica tiene muchas virtudes, principalmente que nos permite dar estimaciones generales de los tiempos de cómputo, pero a la vez presenta algunos problemas importantes, de los cuales quiero mencionar un par: Primero, al eliminar las constantes estamos perdiendo información muy valiosa, finalmente la diferencia entre un segundo, un minuto y un año es constante y, segundo: La información provista por la expresión $f = O(g)$, sólo nos está diciendo que a partir de cierta constante N, $f(n)$, es menor que $cg(n)$, pero esto puede ocurrir para un valor de N enorme, en la práctica puede ocurrir que el punto de intersección entre estas dos funciones (si es que existe) suceda para un valor de N muy grande, es decir, para inputs más grandes que el orden de tamaños para los que nos interese implementar una solución.

Otro problema de manejar este tipo de notación es que muchas veces es extremadamente difícil de calcular el número de pasos de tiempo constante que toma un proceso. Por ejemplo, en algoritmos que se pretenden realizar en computadoras cuánticas, calcular la complejidad computacional puede ser tan difícil que se prefiere usar relaciones de dualidad con teorías relativistas [32]. Para muchos algoritmos (en computación clásica) no es necesario llegar a soluciones tan creativas, basta con hacer pruebas del tiempo que toma un algoritmo con diversos tamaños de input y después ajustar alguna función a los datos recabados, recordando, por supuesto que lo que se quiere es tener una forma

de estimar *cómo crece el tiempo de cómputo* con el input, más que profundizar en los detalles. Este es un procedimiento que se realizará en algunos de los algoritmos que se desarrollaron en este trabajo de investigación, para estimar el tiempo que tomarían en devolver un output, dados los tamaños de input que aquí nos interesan.

3.2. Algoritmos

En la sección anterior escribimos y analizamos la complejidad computacional de un algoritmo para obtener los primeros n elementos de la sucesión de Fibonacci. Para describir a este algoritmo, simplemente propusimos hacer en una computadora, lo que indica la definición para resolver el problema. En general a este procedimiento de proponer un algoritmo simple sugiriendo cómo resolver un problema de una forma directa, le llamamos "proponer un algoritmo ingenuo" (en inglés *naive algorithm*), pocas veces los algoritmos ingenuos son eficientes, o en todo caso no corresponden la forma más eficiente de resolver un problema. Una pregunta que siempre nos haremos para diseñar un algoritmo es: *¿Es este el algoritmo más eficiente?*, siempre que sea posible buscaremos un algoritmo para el que podamos responder esta pregunta afirmativamente (al menos para el tamaño de inputs con los que pensemos estar trabajando) o dar una justificación de por qué para el problema en específico no es posible establecer una solución óptima.

Aunque no hay una forma estandarizada de diseñar un algoritmo (menos uno óptimo) para cualquier problema computacional, existen una serie de técnicas estándar en el mundo del diseño de algoritmos, en esta sección se pretende realizar una descripción de estas metodologías y dar algunos ejemplos de cómo nos ayudan a proponer soluciones que cada vez toman menos tiempo, asintóticamente hablando. Una nota importante que debe quedar clara antes de pasar a los detalles de estas técnicas es que en muchas ocasiones no son suficientes si no se tiene un conocimiento profundo sobre el problema en que se quiere resolver y que, en muchas ocasiones, la propuesta de alguna solución óptima a un problema, no proviene de ellas sino de alguna propiedad interesante del problema, una propiedad matemática o física, por ejemplo.

3.2.1. Técnicas de diseño de algoritmos

Las técnicas de diseño de algoritmo sobre las que hablaremos en esta sección son el diseño codicioso, "divide y vencerás" y programación dinámica. Daremos los detalles de cada una de ellas así como sus ventajas y limitaciones. Así mismo daremos al menos un ejemplo de aplicación de cada una. La primer técnica de la que hablaremos es el diseño codicioso de algoritmos.

3.2.1.1. Algoritmos codiciosos

Un algoritmo codicioso realiza operaciones localmente óptimas en cada paso, esperando que esto resulte en una solución óptima globalmente [10].

Para diseñar un algoritmo codicioso se debe seguir la siguiente estrategia [11]:

1. Hacer una elección codiciosa
2. Reducir el problema original a uno más pequeño
3. Iterar hasta que el problema sea trivial

La elección codiciosa que se hace al principio, debe ser un "movimiento seguro" (una operación localmente óptima), es decir que se debe probar que *existe una solución óptima consistente con la primera operación a realizar*.

En general es sencillo proponer una elección codiciosa para resolver un problema computacional, aunque no siempre es sencillo probar que dicha elección es un movimiento seguro.

Los algoritmos codiciosos y los problemas que pueden resolverse utilizando esta técnica se caracterizan por todas o la mayoría de las siguientes propiedades: [7]

- Tenemos un problema para el que buscamos una solución óptima y para construir dicha solución tenemos una lista de posibles candidatos para ser nuestro primer paso codicioso.
- Conforme el algoritmo avanza conservamos otros dos conjuntos, uno que contiene los candidatos que han sido considerados y utilizados y otro que contiene los candidatos que han sido considerados y rechazados.

Los algoritmos codiciosos serán de particular importancia en este trabajo, puesto que muchos de los algoritmos en gráficas que se utilizarán siguieron este enfoque para su diseño, en particular los algoritmos de *elección de rutas*.

A continuación exploraremos otra técnica de diseño de algoritmo, llamada *divide y vencerás*.

3.2.1.2. Divide y vencerás

Divide y vencerás es una técnica de diseño de algoritmos que consiste en descomponer una instancia de un problema en varias subinstancias del mismo problema, resolverlas sucesiva e independientemente y finalmente combinar las soluciones para encontrar la solución de la instancia del problema original [7].

Este procedimiento se puede segmentar en tres pasos [10]:

- **Dividir** un problema en subproblemas *no traslapados* que sean instancias del problema original.
- **Vencer** a los problemas resolviéndolos recursivamente.
- **Combinar** los resultados de las soluciones de todos los subproblemas en la solución del problema original

Resolviendo eficientemente cada uno de estos pasos, es posible diseñar algoritmos con muy buenos tiempos computacionales, sin embargo, se debe tomar a consideración, que dado que se tiene que guardar información de cada paso recursivo, pueden requerir una gran cantidad de memoria.

El cálculo del tiempo de cómputo de los algoritmos diseñados con la técnica *divide y vencerás* consiste en encontrar la relación recursiva de los tiempos de cómputo entre un problema y sus subdivisiones, para después realizar una suma. Existe un teorema que nos permitirá obtener directamente el tiempo asintótico de un algoritmo según la relación recursiva sin necesidad de hacer una suma cada vez, a continuación se formulará el teorema, según aparece en [10].

Teorema 3.1 (Teorema maestro)**.** *Sean $a \geq 1$ y $b < 1$ constantes y $f(n)$ una función. Sea $T(n)$ definida en los naturales como la relación de recurrencia:*

$$T(n) = aT(n/b) + f(n)$$

donde se puede interpretar a n/b tanto como $\lfloor n/b \rfloor$ o como $\lceil n/b \rceil$. Entonces $T(n)$ tiene las siguientes cotas asintóticas:

1. *Si $f(n) = O(n^{\log_b a - \epsilon})$ para alguna constante $\epsilon > 0$, entonces $T(n) = \Theta(n^{\log_b a})$.*
2. *Si $f(n) = \Theta(n^{\log_b a})$, entonces $T(n) = \Theta(n^{\log_b a} \log n)$*
3. *Si $f(n) = \Omega(n^{\log_b a + \epsilon})$, para alguna constante $\epsilon > 0$ y si $af(n/b) \leq cf(n)$ para alguna constante $c < 1$ y n suficientemente grande, entonces $T(n) = \Theta(f(n))$*

3.2.1.3. Programación dinámica

Esta subsección está basada en una revisión de [13] y [7]. La técnica de programación dinámica está basada en un concepto similar al divide y vencerás. Así como el antes mencionado sugiere dividir un problema en subproblemas sin intersecciones, para resolverlos por separado y luego juntar las soluciones en una solución general. En este enfoque la idea es evaluar un problema recursivamente, considerando subinstancias con posibles superposiciones entre sí. Este enfoque suena extraño si pensamos que cada uno de los subproblemas será resuelto independientemente pero no es así. La idea es mantener una tabla de resultados ya conocidos para evitar cálculos repetidos.

En [7] se menciona que, mientras que la técnica de "divide y vencerás" comienza del problema más general y va subdividiendo en componentes cada vez menores, la programación dinámica va construyendo desde lo más pequeño, combinando soluciones hasta que llegamos al problema original.

En [13] se da otro ejemplo relacionado con un tema en el que ahondaremos posteriormente, calcular el camino más corto entre todos los pares de nodos de una gráfica:

El camino más corto entre todos los pares de nodos: Imaginemos que ya tenemos una manera de calcular el camino más corto entre dos nodos u, v de una digráfica $D = (V, A)$, llamémosle al algoritmo que calcula este camino `D(u,v)`. Como se verá en la sección 4.1, este algoritmo posiblemente tiene que recorrer distintos caminos posibles entre los nodos u y v y luego determinar cuál es el menor de ellos. Si quisiéramos calcular el trayecto más corto entre todas las parejas de nodos u, v a través de llamar repetidamente `D(u,v)`, seguramente analizaríamos cada arista como opción muchas veces y nuestro cálculo se volvería muy ineficiente. Una opción en vez de ello es el *algoritmo de Floyd-Warshall*, en el cual se hace uso de una matriz de $n \times n$ donde $n = |V|$ donde el elemento $D_{i,j}$ es la distancia más corta entre el nodo i y el nodo j. El algoritmo va construyendo una secuencia de matrices D^0, D^1, ..., D^k, donde D^0 es simplemente la matriz de adyacencias y D^k contiene las distancias más cortas con la condición de que sólo se consideran los trayectos en los cuales los nodos intermedios no tienen un índice mayor a k. Resulta que:

$$D^k_{i,j} = min\left\{D^{k-1}_{i,j}, D^{k-1}_{i,k} + D^{k-1}_{k,j}\right\}$$

puesto que todos los caminos óptimos deben estar compuestos de subcaminos óptimos, ya que de otra forma podríamos substituir el subcamino encontrando un camino mejor.

Hasta ahora ya hemos hablado de diferentes aproximaciones para implementar un algoritmo, a continuación iremos a un tema interesante que puede llegar a ser hasta contradictorio con las buenas prácticas que hemos establecido: En ocasiones es mejor implementar un algoritmo incorrecto o que sólo nos entrega una respuesta aproximada.

3.2.2. Algoritmos incorrectos

¿Por qué querríamos utilizar un algoritmo si no podemos tener la certeza de que nos darán la respuesta correcta? Esto sólo tiene sentido si algún algoritmo que sí nos aseguraría esto tiene alguna desventaja importante. En la siguiente sección se hablará sobre un tipo de problemas que tienen esta característica y posteriormente se hablará de cómo, para este tipo de problemas, se utilizan aproximaciones interesantes, que si bien no nos garantizan una respuesta exacta, nos garantizan una aceptable.

3.2.2.1. Problemas NP-Completos

Los algoritmos para los cuales la complejidad es $O(n^k)$ donde n es el tamaño del input y k alguna constante, son llamados **problemas solubles en tiempo polinomial**. Se podría pensar que para cualquier algoritmo se puede encontrar alguna k que limita la complejidad computacional pero no es así, hay problemas que no pueden ser resueltos en tiempo polinomial.

En general es posible pensar a los problemas solubles en tiempo polinomial como problemas más sencillos y cuya solución es menos costosa y a los problemas no solubles en tiempo mayor al polinomial como problemas más complejos y con soluciones más costosas. Pero aún hay otro nivel de complejidad, existen algoritmos conocidos como **NP-Completos** para los cuales ni si quiera se sabe cuál es su complejidad computacional. Para estos problemas no se ha descubierto ninguna solución en tiempo polinomial, pero tampoco se ha probado que no son solubles en tiempo polinomial [10].

A continuación se darán algunas definiciones, relacionadas con solubilidad en tiempo polinomial, estas definiciones fueron extraídas de un estudio de [10] y [7]:

- Un problema es **de clase P** si es posible resolverlo en tiempo polinomial.
- Un problema es **de clase NP** si es posible verificar que una posible solución del problema es efectivamente una solución del problema en tiempo polinomial.

Nótese que todos los problemas de clase P son de clase NP, porque para esos problemas podemos simplemente encontrar la solución en tiempo polinomial y saber que esa solución es correcta. La pregunta de si el caso contrario es cierto es un problema abierto.

Utilizando estas definiciones, podemos definir a los problemas **NP-Completos** con algo más de formalidad: Los problemas NP-Completos son problemas de tipo NP y que son tan complicados como cualquier problema que también es de tipo NP, si se desea saber a qué se refiere exactamente la frase "tan complicado como cualquier problema de NP" es mejor referirse a [10] o a [7], puesto que es necesario una serie explicaciones sobre cómo transformar un problema de optimización en un problema de decisión (con respuesta sí o no) y de cómo comparar problemas de decisión.

Para los objetivos de este trabajo nos limitaremos a decir que algo interesante de este tipo de problemas (NP-Completos) resulta que es posible probar que todos ellos tienen complejidades equivalentes y más aún, se sabe que si se descubriera algún algoritmo capaz de resolver **un** problema NP-Completo, automáticamente se tendrían soluciones para **todos** los problemas NP-Completos [7]. Y que si es posible probar que un problema es NP-Completo se puede tener una gran certeza de que el problema es difícil, de que al momento nadie sabe como resolverlo eficientemente y de que es mucho mejor buscar un algoritmo aproximado o resolver un caso particular que sea menos difícil.

Algunos problemas NP-Completos en el área de la teoría de gráficas son:

1. El problema del agente viajero (o TSP por sus siglas en inglés): Dado un conjunto de ciudades y la distancia entre cada par de ciudades, encontrar la ruta más corta posible que visita todas las ciudades una sola vez y que regresa a la ciudad original.
2. El problema de ciclo Hamiltoniano: Dada una gráfica no dirigida, encontrar un ciclo que visita cada vértice una sola vez. Este problema es similar a TSP pero sin la necesidad de minimizar la distancia.

3. El problema de encontrar el camino simple más largo en una gráfica.

Para finalizar, esta sección fue incluida precisamente porque la asignación de rutas y la predicción de flujos vehiculares en redes son también problemas NP-Completos ([1], [20], [5]). Por ello, exploraremos soluciones aproximadas para las simulaciones que requerimos para nuestro tema de estudio.

3.2.2.2. Algoritmos aproximados y heurísticas

Existen al menos tres maneras de resolver un problema aún si es muy complejo (incluso si es NP-completo). La primera es si los inputs son muy pequeños, en ese caso, aún si la complejidad es exponencial, posiblemente podamos encontrar una solución en un tiempo razonable. La segunda es si aislamos un caso particular del problema y resolvemos sólo para esa instancia. La tercera es encontrar una aproximación, una solución cercana al óptimo. Este tipo de soluciones es de las que hablaremos en este capítulo [10].

A continuación se dan dos definiciones importantes: El término **algoritmo aproximado** se utiliza para un procedimiento que siempre obtiene algún tipo de solución para el problema, aunque pueda fallar en encontrar una solución óptima. Por otro lado una **heurística** es un procedimiento que puede producir una solución buena o inclusive óptima al problema en algunos casos, pero que sin embargo puede no producir ninguna solución o alguna que esté muy lejos de la optimalidad. [7].

Todas las técnicas que fueron descritas en este capítulo pueden ser usadas también para desarrollar un algoritmo aproximado, o una heurística, la principal diferencia es que para estos casos no es necesario garantizar que se llegará a la solución óptima, sino que simplemente, en la mayoría de los casos, se obtendrá una solución aceptable. A continuación se da un ejemplo de coloración en los vértices de una gráfica.

Coloración de los nodos de una gráfica: Sea $G = (V, A)$ una gráfica no dirigida. La meta es colorear los nodos de G (dar un mapeo de índices a colores) de una forma tal que no haya dos nodos adyacentes coloreados del mismo color. En general se trata de resolver esto con el mínimo número de colores posible (al cuál se le llama el *número cromático de* G). Encontrar el número cromático de una gráfica es NP-Completo.

Una solución heurística codiciosa consiste en escoger un color c_1 y un nodo $v \in V$ arbitrario, después se consideran todos los demás nodos y simplemente se sigue la regla de que: Si el nodo puede ser coloreado con c_1 (ninguno de sus vecinos ha sido coloreado con c_1) se asigna ese color, en caso contrario nos lo saltamos hasta el siguiente paso del algoritmo. Una vez que todos los nodos que pudieron ser pintados con c_1 están listos, tomamos otro color c_2 y un nodo aún no coloreado, para después tratar de colorear a todos los nodos aún no coloreados con c_2, siguiendo la regla de que ningún vecino puede haber sido coloreado con c_2. Si aún hay nodos sin colorear, se elige un color c_3 y se repite la misma lógica, se continua hasta que todos los nodos hayan sido coloreados con algún color.

Para cualquier gráfica G existe al menos un ordenamiento de los nodos donde este algoritmo nos dará una coloración con el número de colores siendo el número cromático

de G, pero habrá muchos ordenamientos donde esto no es verdad. También hay gráficas para las cuales el número de colores obtenido puede estar extremadamente alejado del número crómatico si uno tiene mala suerte al escoger el nodo inicial.

Este algoritmo es una heurística y lamentablemente no hay ninguna forma de acotar el error de la solución. Una forma de protegerse es corriendo el algoritmo en múltiples ocasiones y guardando el resultado mínimo. para un número suficientemente grande de corridas tendremos una idea bastante buena de cuál es el número cromático de una gráfica, en el peor de los casos, cuando la gráfica es completa, el tiempo para correr todas las opciones posibles es $O(n^3)$, lo que puede ser razonable según el tamaño de la gráfica de la que hablemos.

3.3. Estructuras de datos

En esta sección se mencionarán algunas estructuras de dato que serán mencionadas en este trabajo o que fueron utilizadas en la implementación de las simulaciones. Las definiciones presentadas en las siguientes subsecciones fueron redactadas a partir de una revisión de [6], de [7] y de [25].

3.3.1. Arreglos y *linked lists*

Los arreglos son una estructura de datos que consiste en un grupo de elementos del mismo tipo, por ejemplo representaciones de reales (números de punto flotante) o enteros. En la memoria de una computadora la propiedad que tienen los arreglos es que almacenan los elementos en celdas contiguas y por tanto es posible acceder a cada uno de ellos indicando un índice de forma tal que la lectura del elemento almacenado en el índice i o incluso cambiar un valor almacenado en el índice i toma un tiempo constante $O(1)$. Por otro lado, cualquier operación que requiera recorrer todos los elementos tomará un tiempo $O(n)$ siendo n el número de elementos en el arreglo.

Las *linked lists*, o simplemente "listas"son un tipo de extensión de un arreglo en el cual el número de elementos no está determinado de antemano. De esta estructura se pide saber cuál es el primer elemento, cuál es el último y para cualquier elemento, quienes son su antecesor y sucesor si estos existen.

A continuación se explicarán dos tipos de formas de crear un arreglo o una linked list, que permiten optimizar algunas operaciones según para lo que se desee utilizar la estructura de datos.

3.3.1.1. *Stacks* (pilas)

Los *stacks* son descritos como estructuras de tipo LIFO (*Last In First Out*), lo que literalmente quiere decir que el último elemento en ser agregado es el primero que es extraído. La razón por la que esta estructura es llamada "*stack*" o "pila", es porque una analogía comúnmente utilizada para comprender una estructura LIFO es la de una

pila, por ejemplo una pila de platos sucios por lavar. Siempre que se agrega un nuevo plato, este se coloca encima de los demás, convirtiéndose en el plato superior. Siempre que se retira un plato, se toma el de más arriba, es decir, el que fue agregado más recientemente.

Las pilas son implementadas con un arreglo o una lista (si es que no se desea determinar desde un principio el número máximo de elementos) cuyos elementos están indexados con un índice que corre de 1 al tamaño máximo requerido n, junto con un contador. Vaciar una pila significa poner el contador en cero, después, para añadir un nuevo elemento, el contador se incrementa en 1 y se guarda al elemento deseado en la posición índice = contador. Para remover un elemento de la pila se lee el último que fuera agregado, es decir, el que tiene índice correspondiente a índice = contador y posteriormente se decrementa al contador en 1. A la operación de incluir un elemento en un stack comúnmente se le llama `push`, mientras que la operación para remover a un elemento recibe el nombre de `pop`.

3.3.1.2. *Queues* (colas)

Las *queues*, filas o colas son descritas como estructuras de tipo FIFO (*First In First Out*), lo que literalmente quiere decir que el elemento que llevan más tiempo en la estructura (o bien los primeros en ser agregados) son los primeros en salir. A estos elementos se les llama queues, filas o colas, porque recuerdan a lo que debe pasar en una fila, la primer persona en formarse es la primera que es atendida.

La implementación de una cola es un poco más complicada que la de una pila porque los cambios deben ocurrir en ambos extremos, mientras que en un lado se insertan elementos, del otro se extraen. Para ver implementaciones explícitas, se sugiere revisar el capítulo 1 de [6].

3.3.2. Árboles

Un árbol es una gráfica conexa y acíclica, como se vio en la sección 1.1.8. Un nodo del árbol puede ser determinado como su "raíz", en el árbol entendido como estructura de datos, la raíz es el punto de partida de las exploraciones o búsquedas sobre el mismo y a partir de la raíz es posible entender la relación entre nodos adyacentes como de parentesco, siendo el padre el nodo más cercano a la raíz.

Lo interesante de los árboles como estructura de datos es que en general cada nodo tiene un valor guardado y que mantiene un orden entre los objetos almacenados. La forma en que es posible guardar dicho orden es a través de las relaciones entre los objetos, que usualmente consisten en apuntadores que van de cada nodo hacia todos sus nodos hijos, en general es aceptado que el orden de los hijos se especifica de izquierda a derecha.

Un árbol es llamado binario si cada nodo tiene a lo más dos hijos.

3.3.3. *Heaps* o filas de prioridad

Los *heaps* o filas de prioridad son estructuras que mantienen un conjunto de elementos asociados con un valor (su prioridad). Pueden ser entendidos como un tipo especial de árbol enraizado que puede ser implementado en un arreglo sin ningún apuntador en específico. La diferencia principal entre un *heap* y un árbol de búsqueda es que los *heaps* están diseñados para poder seleccionar el elemento con mínima o máxima prioridad en vez de realizar búsquedas.

En general se utilizan *heaps* binarios, que son un tipo particular de árbol binario en el que se cumplen dos propiedades clave:

- Propiedad de orden parcial: En un *heap* binario, para cualquier nodo v con un padre u, el valor almacenado en u es mayor o igual que el valor almacenado en v (si es un *heap* máximo) o menor o igual (si es un *heap* mínimo).
- Propiedad de árbol completo: Un *heap* binario es un árbol binario completo, lo que significa que todos los niveles del árbol están completamente llenos, excepto posiblemente el último nivel, que se llena de izquierda a derecha.

Los *heaps* se utilizan principalmente para implementar colas de prioridad, donde los elementos se organizan en función de su prioridad. En un *heap* máximo, el elemento de mayor prioridad se encuentra en la raíz del árbol, mientras que en un *heap* mínimo, el elemento de menor prioridad se encuentra en dicha posición.

Las operaciones básicas que se pueden realizar en un *heap* incluyen la inserción de elementos, la eliminación del elemento de mayor prioridad (o menor prioridad, según sea el caso) y la obtención del elemento de mayor prioridad (o menor prioridad) sin eliminarlo.

Capítulo 4

Algoritmos en gráficas

Para lograr simular la forma en que los conductores toman decisiones en la ciudad, partimos de la idea que siempre tratarán de elegir la ruta que les permita llegar de su origen a su destino en el menor tiempo posible. Es por esto que en esta sección se analizarán algoritmos de búsqueda en gráficas, en este caso particular *la búsqueda de la ruta que minimiza el tiempo.*

4.1. Algoritmos de búsqueda en gráficas

Esta sección se escribió partiendo de una revisión bibliográfica de [27] y [13], en particular las piezas de pseudocódigo presentadas en este capítulo, son una adaptación de [13].

Los algoritmos de búsqueda pueden ser clasificados en dos categorías: Los algoritmos no informados, que son aquellos que no tienen información del problema más allá de su definición, y los algoritmos informados, o que integran heurísticas, que tienen cierta información acerca de una buena forma de buscar soluciones.

4.1.1. Algoritmos no informados

Los algoritmos no informados no contienen ninguna información previa del problema, además de su definición y la capacidad de distinguir el momento en que han alcanzado el nodo destino. En general haremos *Búsquedas implícitas*, lo que quiere decir que no hay una representación inicial de la estructura de la gráfica, sino que poco a poco se irán construyendo imágenes parciales de las posibles trayectorias a partir de un nodo conforme el algoritmo avanza y más nodos son explorados. En este tipo de procesos se comienza con un nodo inicial, el origen, y a partir de él se van explorando sus nodos vecinos, y así sucesivamente, hasta que se alcanza el nodo destino.

Los nodos de la gráfica siendo explorada pueden ser clasificados según si ya han sido visitados por el algoritmo o aún no. Y entre los nodos que ya han sido alcanzados por

el algoritmo podemos hacer una subclasificación según si ya se han explorado todos sus vecinos o no. Esta clasificación puede visualizarse en el siguiente esquema (4.1):

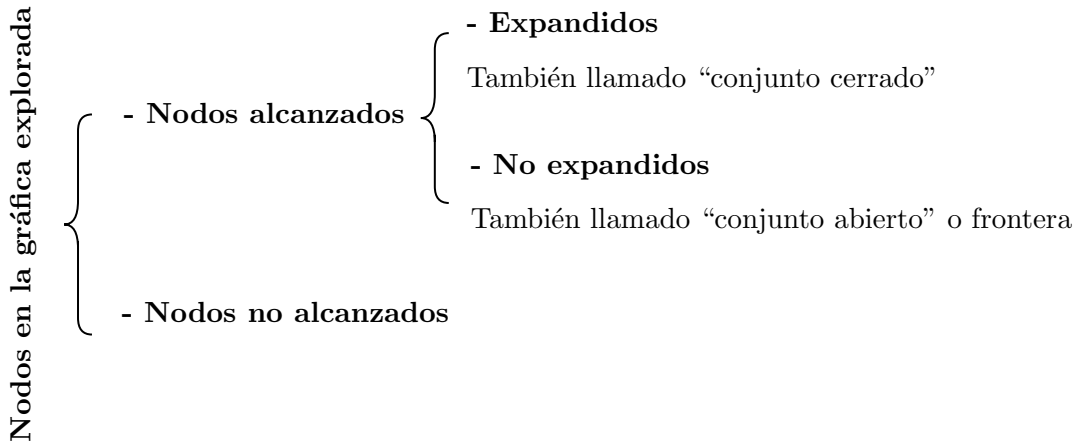

Figura 4.1: Categorización de los nodos de una gráfica que está siendo explorada

Este procedimiento se puede abstraer como la construcción secuencial de un árbol, cuya raíz es el nodo inicial y que contiene los caminos ya expandidos a partir de él en la gráfica del problema. Este árbol, conocido como "árbol de búsqueda" caracteriza la parte de la gráfica que ya ha sido explorada por el algoritmo en un instante del tiempo y sus hojas corresponden al conjunto abierto o frontera de búsqueda.

En este árbol puede haber nodos "repetidos" en distintas secciones, si es que el algoritmo de búsqueda llega a un nodo más de una vez, sin embargo, se hace una distinción importante entre los nodos de la gráfica original (a los que a partir de ahora llamaremos *estados*)y los nodos del árbol de búsqueda (a los que seguiremos llamando *nodos*), que contienen no sólo información de la posición en la que nos encontramos en la gráfica original, sino también de cómo accedimos a ella, información que usualmente se guarda como un apuntador al vértice antecesor. Esta distinción nos permite llamar a la estructura donde guardamos la información de la sección ya explorada de la gráfica "árbol", ya que al no haber dos nodos idénticos (aunque compartan el mismo *estado*), no se forman ciclos.

En la figura 4.2 se muestra un ejemplo concreto del árbol de búsqueda en un momento dado de la aplicación del algoritmo sobre una gráfica. Las hojas de dicho árbol (c,d) son los elementos del conjunto abierto, los nodos a,b y f, son los elementos del conjunto cerrado, mientras que el estado e no ha sido explorado.

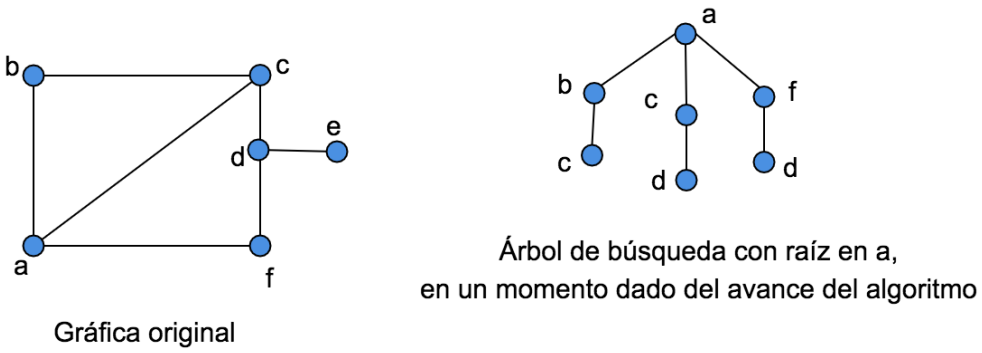

Figura 4.2: Ejemplo de la generación de un árbol de búsqueda a partir del proceso de exploración de una gráfica.

El algoritmo general a través del cual se va explorando la gráfica es el siguiente: Mientras no haya una solución (un camino entre el origen y el destino o bien la certeza de que no existe tal camino) un nodo del conjunto abierto se selecciona y se genera una lista con todos sus sucesores. Para cada uno de los sucesores se aplica una subrutina `Mejora`, que a su vez actualiza a los conjuntos abierto y cerrado, estos conjuntos deben ser implementados con una estructura de datos que permita una fácil remoción e inserción de elementos.

Algo que se puede notar en el árbol de la figura 4.2 es que hay varios estados repetidos, ya vimos que esto no afecta la propiedad acíclica de dicho árbol, sin embargo, aún para una gráfica tan pequeña como la de esta figura nos podríamos enfrentar al problema de tener una árbol de búsqueda infinito si no lidiamos con los elementos repetidos del conjunto cerrado, puesto que los nodos podrían seguir siendo expandidos y expandidos, a pesar de que ese estado ya se haya considerado con anterioridad. Para evitar que este fenómeno suceda, provocando que el algoritmo nunca converja, es necesario encontrar una forma de detectar estados duplicados y poner alguna regla operativa de cómo hacer para eliminar dichos estados.

Una forma de hacer esto es manteniendo el nodo en el conjunto y simplemente actualizando un apuntador hacia su antecesor cuando sea necesario. La forma específica de hacer esta actualización, así como la regla de elección de qué nodo del conjunto abierto debe ser expandido en cada paso son las dos características que diferencian a los diferentes algoritmos no informados. Antes de comenzar a describirlos individualmente, se presenta el procedimiento general para toda esta familia de algoritmos, llamado "búsqueda implícita".

Algoritmo 2: Búsqueda implícita

Datos: Una gráfica G, un nodo inicial s, pesos en las aristas w, una función para generar sucesores `Expandir` y un nodo destino t.
Resultado: Un camino de s a t, o $\emptyset$ en caso de que dicho camino no exista.

```
Cerrado ← ∅
Abierto ← {s}
mientras Abierto ≠ ∅ hacer
    Retirar algún nodo u de Abierto
    Añadir u en Cerrado
    si u = t entonces
        devolver camino desde s hasta u (llamado Camino (u))
    fin
    sucesores(u) = Expandir (u)
    para cada nodo v en sucesores(u) hacer Mejora (u,v);
fin
devolver ∅
```

La forma de obtener el camino de s a u es mediante un proceso de seguir los apuntadores hacia los antecesores de cada nodo según el siguiente algoritmo:

Algoritmo 3: Camino

Datos: Nodo u y nodo inicial s de una gráfica G con función de pesos en las aristas w y un conjunto de apundarores de los nodos a sus antecesores (invecinos), dado por el algoritmo de búsqueda.
Resultado: Un camino de s a u.

```
Camino ← (u)
mientras antecesor(u) ≠ s hacer
    Camino ← (u, Camino)
    u ← antecesor(u)
fin
devolver (s, Camino)
```

Como se mencionó con anterioridad, una forma de lograr la eliminación de duplicados en el conjunto cerrado es mediante una actualización de los apuntadores hacia el padre (o antecesor) de los nodos. Esto se puede ejemplificar con la siguiente función `Mejora` (una versión sencilla del procedimiento, que no asegura un camino de costo mínimo, sino simplemente un camino desde s hasta u).

Algoritmo 4: Mejora

Datos: Dos nodos u y v tales que v es sucesor de u.
Efecto: Actualiza el valor del antecesor de v y modifica los conjuntos *Abierto* y *Cerrado*.
si $v \notin Abierto \cup Cerrado$ **entonces**
    Insertar v en *Abierto*
    $antecesor(v) \leftarrow u$
fin

En este algoritmo, no se habla de un resultado, sino de un efecto, puesto que no se obtiene ningún valor de salida, sino que se producen cambios sobre las estructuras de datos del problema.

Este pseudocódigo es útil para cualquiera de los algoritmos no informados de los que hablaremos a continuación, así como la función `Camino`. Por otro lado, la función `Mejora`, presentada aquí es un ejemplo inicial, pero puede variar entre algoritmos, así como la función `Expandir` que, puesto que es la función que describe cuál es el siguiente nodo a ser expandido, es de hecho el punto central en el que radica la diferencia entre los algoritmos de búsqueda no informada.

4.1.1.1. *Breadth first search* (BFS o Amplitud primero)

Para los algoritmos tipo **amplitud primero**. El conjunto `abierto` se implementa en una estructura tipo *FIFO* (*First In First Out*) Esto quiere decir, el elemento más antiguamente insertado en la estructura es el primero en ser eliminado, para tener más detalles de la estructura de datos, se puede consultar la sección 3.3.1.2. La estructura tipo FIFO asegura que se recorra en amplitud primero ya que los nuevos nodos (que siempre son más profundos que sus padres) se colocan al final de la cola, y los nodos antiguos, que son más superficiales que los nuevos nodos, se expanden primero.

BFS siempre encontrará el camino menos profundo hacia cualquier nodo en la fornтera, sin embargo esto no asegura que sea el camino óptimo, esto únicamente sucederá si el costo de un camino siempre es una función no decreciente de la profundidad. Por ejemplo si todas las aristas tienen el mismo peso.

Un posible problema para los algoritmos BFS es que su complejidad computacional en tiempo y en espacio no es muy buena: En un árbol uniforme en el que en cada estado hay b sucesores, la raíz genera b nodos en el primer nivel y posteriormente cada nodo a su vez genera b nodos más, siguiendo esta lógica, en el nivel n habría b^n nodos, y a la profundidad p del árbol: b^p. Por lo que la complejidad en tiempo es $O(b^d)$.

Para la complejidad espacial, dado que cada nodo generado permanece en la memoria, al final habrá $O(b^{d-1})$ nodos en el conjunto explorado y $O(b^d)$ en la frontera.

4.1.1.2. *Depth first search* (DFS o Profundidad primero)

Para los algoritmos tipo **profundidad primero**. El conjunto `abierto` se implementa en una estructura tipo *LIFO* (*Last In First Out*) Esto quiere decir, el elemento más reciente que se inserta en la estructura es el primero en ser eliminado (para revisar la implementación de esta estructura consultar la sección 3.3.1.1).

La razón por la que esta estructura asegura una búsqueda en profundidad primero es que siempre se explora tan lejos como sea posible a lo largo de cada rama antes de retroceder, ya que cuando se llega a un nodo y se encuentran sus sucesores, estos se agregan a la pila en el orden en el que deben ser explorados. El algoritmo comienza en un nodo inicial y se adentra en el grafo siguiendo una ruta hasta que alcanza un nodo sin nodos sucesores no visitados . En ese punto, retrocede al nodo anterior y continúa explorando otras ramas no exploradas.

La complejidad computacional de DFS en tiempo puede ser igual o peor que la de BFS, sin embargo la complejidad espacial (de memoria) en general es mucho ya cada vez que un nodo ha sido expandido, puede ser removido de la memoria siempre que sus exvecinos hayan sido explorados.

4.1.1.3. Algoritmo de Dijkstra

El algoritmo de Dijkstra permite trabajar con aristas que tienen un peso dado por una función de costo y hace uso de la información que tenemos de dichos pesos. Es por ello que este algoritmo no usa estructuras tipo LIFO ni FIFO para el conjunto `abierto`, en vez de eso utiliza una estructura llamada *priority queue* o "fila de prioridades"(3.3.3), en la cual se asocia a cada nodo con el costo que toma alcanzarlo, de forma tal que se explora la red a través de caminos de menor costo .

Más especificamente, Dijkstra propone que el camino de menor costo del nodo origen s a un nodo v $(\delta(s,v))$es el mínimo de la suma del costo de alcanzar u $(\delta(s,u))$más el costo de la arista (u,v) $(w(u,v))$. Esto quiere decir que cualquier subcamino de un camino óptimo, también es óptimo ya que, de otra forma, podríamos reemplazar el subcamino por otro mejor y optimizar también así el camino principal (de esto ya habíamos hablado en uno de los ejemplos de la sección 3.2.1.3).

En un principio el algoritmo de Dijkstra mantiene una cota superior de costo para cada nodo, conforme se van explorando los nodos este valor se corrige hasta alcanzar el mínimo costo posible para alcanzarlo.

La función `Mejora` del algoritmo de Dijkstra se ve así:

Algoritmo 5: Mejora (Dijkstra)

Datos: Dos nodos u y v tales que v es sucesor de u.
Efecto: Actualiza el valor del antecesor de v y modifica los conjuntos *Abierto* y *Cerrado*.

```
si v ∈ Abierto entonces
    si f(u) + w(u + v) < f(v) entonces
        antecesor(v) ← u
        Se actualiza f(v) como f(u) + w(u + v)
    fin
en otro caso
    si v ∉ Cerrado entonces
        antecesor(v) ← u
        Se inicializa f(v) como f(u) + w(u + v)
        Se inserta v en Abierto con f(v)
    fin
fin
```

Donde $f(u)$ es la estimación de costo de alcanzar el nodo u desde s.

El algoritmo comienza con el nodo origen s con $f(s) = 0$ (lo que significa que no tiene ningún costo desplazarse de s a s) e inicializa $f(v)$ para todos los nodos en la gráfica con un valor de ∞. Posteriormente, en cada iteración se elige al nodo u de menor de menor costo y se considera a todos sus exvecinos $v \in N_D^+(u)$, para después utilizar la función `Mejora` para inicializar o actualizar el estimado de $f(v)$ de todos los $v \in N_D^+(u)$. Siempre que sea posible encontrar un camino de menor costo entre s y v, se elige dicha opción.

El algoritmo continúa seleccionando y visitando nodos hasta que se hayan visitado todos los nodos o se alcance el nodo de destino. Al final, se obtiene el camino más corto desde el nodo de origen a todos los demás nodos en el grafo o bien el camino más corto hasta el nodo destino.

Es posible probar para el algoritmo que:

- Si $G = (V, A, w)$ es una gráfica con pesos positivos y $f(v)$ es la aproximación de $\delta(s, u)$ en el algoritmo de Dijkstra. Al momento en que u es seleccionado para ser expandido, se tiene que $f(u) = \delta(s, u)$. Esto no es cierto si los pesos no son estrictamente no negativos.

- Si $G = (V, A, w)$ es una gráfica con pesos positivos, el algoritmo de Dijkstra es óptimo. Es decir que en cuanto se selecciona el nodo destino t para ser expandido, sucede que $f(t) = \delta(s, t)$

- La complejidad estimada del algoritmo es $O((V + E)logV)$ donde V es el número de vértices y E el número de aristas. Siempre y cuando la implementación de la fila de prioridad sea realizada correctamente. En otro caso la complejidad puede aumentar hasta $O(V^2 + E)$

Es posible encontrar otros algoritmos que permitan explorar gráficas con pesos negativos, sin embargo, no exploraremos dichos algoritmos, puesto que los pesos con los que se trabajó en el desarrollo de este proyecto (tiempos, velocidades y distancias) son siempre positivos. En vez de eso hablaremos de otro tipo de algoritmos que permiten hacer búsquedas de forma más eficiente, ya que integran información adicional sobre el problema y por tanto en muchos casos se ahorran el expandir nodos innecesarios.

4.1.2. Algoritmos que integran heurísticas

La otra categoría de algoritmos de búsqueda en gráficas, corresponde a los algoritmos que sí tienen información adicional del problema, específicamente una heurística.

4.1.2.1. Heurísticas

Las heurísticas en este contexto son **estimados** de la distancia (o costo) restante para alcanzar un nodo destino d a partir de otro. La información se incluye en los algoritmos de búsqueda para establecer si hay un posible estado que se vea más prometedor y que por tanto es mejor explorar primero. Una heurística es esencialmente una función de los nodos a los reales positivos, ya que sólo trabajaremos con aristas con pesos positivos. Dichas funciones pueden tener las siguientes propiedades:

Si $G = (V, A)$ es una gráfica con pesos w en las aristas y $h(v)$ una heurística:

- Admisibilidad: $h(v)$ es admisible si es una cota inferior de los costos de la solución óptima, es decir que nunca sobreestima el costo o la distancia, o bien $h(u) \leq \delta(u, d)$ para un nodo destino d y cualquier nodo $u \in V$.
- Consistencia: $h(v)$ es consistente si $h(u) \leq h(v) + w(u, v)$ para cualquier arista $(u, v) \in A$
- Monotonicidad: Sea $(u_0, ..., u_k)$ una trayectoria en G, $g(u_i)$ el costo de una subtrayectoria $(u_0, ..., u_i)$ y $f(u_i) = g(u_i) + h(u_i)$. $h(v)$ es monótona si $f(u_j) \geq f(u_i)$ para cualquier par i, j tal que $0 \leq i < j \leq k$. O sea que el estimado de un camino total es no decreciente desde un nodo hacia sus sucesores.

Se puede demostrar que las dos últimas son equivalentes y que la consistencia implica admisibilidad.

A continuación se describirán un par de algoritmos que utilizan este tipo de funciones con la finalidad de reducir la complejidad computacional de la búsqueda.

4.1.2.2. *Greedy best first search*

Este algoritmo trata de expandir primero los nodos más cercanos a la meta, así que evalua a los nodos usando como estimación del costo $f(s)$ de alcanzar el nodo objetivo u desde s, exactamente igual a una heurística: $f(n) = h(s)$. El algoritmo se

llama codicioso justamente porque en cada paso trata de acercarse lo más posible a la meta. En el peor caso posible la complejidad de este algoritmo es $O(b^m)$ donde m es la máxima profundidad en el espacio de búsqueda. En general la complejidad dependerá de qué tan buena sea la heurística elegida, lo que también aplicará para el siguiente algoritmo descrito que es de fundamental importancia para este trabajo.

4.1.2.3. Algoritmo A*

El algoritmo A* evalua a los nodos con una función de costo $f(s)$ compuesta de dos cosas: Primero, utiliza el costo de alcanzar s desde el nodo origen: $g(s)$. Segundo, integrando una heurística del costo de alzanzar la meta desde s: $h(s)$. La función de costo es $f(s) = g(s)+h(s)$, que se puede entender como el costo estimado de la solución más barata (de menor distancia) que pasa por s.

El algoritmo A* se ve así:

Algoritmo 6: A*

Datos: Gráfica de búsqueda implícita con nodo inicial s, nodo final t, función de pesos w, heurística h y una función de generación de sucesores: `Expandir`.
Resultado: Camino de costo óptimo de s a t o $\emptyset$ si no existe dicho camino.

```
Cerrado ← ∅
Abierto ← {s}
f(s) ← h(s)
mientras Open ≠ ∅ hacer
    Remover u de Abierto con el mínimo valor posible de f(u)
    Insertar u en Cerrado
    si u = t entonces
        devolver Camino(u)
    en otro caso
        Sucesores(u) ← Expandir (u)
        para v ∈ Sucesores(u) hacer
            Mejora(A*) (u, v)
        fin
    fin
fin
devolver ∅
```

donde la función `Mejora` puede ser descrita con el siguiente algoritmo:

Algoritmo 7: Mejora (A*)

Datos: Nodos u y v, donde v es un sucesor de u.
Efecto: Actualización del antecesor de v, el costo $f(v)$ y los conjuntos *Abierto* y *Cerrado*

```
si v ∈ Abierto entonces
    si g(u) + w(u,v) < g(v) entonces
        antecesor(v) ← u
        f(v) ← g(u) + w(u,v) + h(v)
    fin
si no, si v ∈ Cerrado entonces
    si g(u) + w(u,v) < g(v) entonces
        antecesor(v) ← u
        f(v) ← g(u) + w(u,v) + h(v)
        Se retira a v de Cerrado
        Se coloca a v en Abierto con peso f(v)
    fin
en otro caso
    antecesor(v) ← u
    Se inicializa f(v) como g(u) + w(u,v) + h(v)
    Se coloca v en Abierto con peso f(v)
fin
```

A* ofrece soluciones óptimas y además lo hace expandiendo el mínimo número de nodos siempre y cuando la heurística usada sea consistente. Si sólo se tiene admisibilidad pero no consistencia es posible que haya nodos que se reabran y de hecho el número de aperturas puede crecer exponencialmente en casos graves (aunque esto no pasa muy frecuentemente en aplicaciones).

La complejidad de A* es exponencial con exponente *el error absoluto* de la heurística (la diferencia entre el costo real y el estimado), cuando esto es una limitante, en general lo que se hace es usar variantes de A* que encuentran soluciones subóptimas o bien diseñar heurísticas que reducen el error aún si no son estrictamente admisibles. En términos de memoria su complejidad también es grande porque requiere de tener en memoria los conjuntos Abierto y Cerrado.

Hay algoritmos que pueden ayudar a resolver los problemas de complejidad espacial y a la vez garantizar optimalidad (por ejemplo el iterative-deepening A* (IDA*)), sin embargo estos quedan fuera del alcance de este proyecto.

4.2. Breve discusión de algoritmos relacionados a flujo en redes

En esta sección se mencionan algunas ideas de los algoritmos existentes para el cálculo de flujo en redes, como se mencionó en la sección de flujos en redes 1.2. Lo escrito en esta sección está basado en una revisión de [2].

4.2.1. Algoritmos para el problema de flujo máximo

Brevemente el problema de flujo máximo es: *En una red con capacidades en las aristas, se desea mandar el máximo flujo posible entre un nodo origen s y un nodo destino t, sin exceder la capacidad de ninguna arista.*

Los algoritmos para resolver este problema son de dos tipos: **Algoritmos que van haciendo crecer poco a poco el flujo sobre una trayectoria** del nodo origen s al nodo destino t y que mantienen la condición de conservación de masa en todos los nodos. Escencialmente:

Algoritmo 8: Augmenting Path

Datos: Gráfica G, nodo origen s y nodo destino t.
Efecto: Actualización de $G(x)$.
$x \leftarrow 0$
mientras $G(x)$ *contiene una trayectoria de* s *a* t **hacer**
  Se identifica una trayectoria P de s a t
  $\delta \leftarrow min\{r_{i,j} | (i,j) \in P\}$
  Se aumenta el flujo en δ unidades a través de P
  Se actualiza $G(x)$
fin

Donde r_{ij} es la capacidad residual de una arista (i,j), es decir la diferencia entre su capacidad y el flujo que corre a través de ella. y $G(x)$ es la gráfica residual respecto al flujo x, es decir la gráfica con la capacidad residual en las aristas que queda después de hacer correr x.

La complejidad de este tipo de algoritmos es cuadrática o lineal, en particular de un caso llamado **algoritmo de etiquetado** la complejidad es de $O(nmU)$ donde n es el número de nodos, m el número de aristas y U un real positivo que funciona como cota para las capacidades de todas las aristas de la red.

Algoritmos llamados "preflow-push", que inundan la red de forma que puede ser que haya varios nodos con más flujo que su demanda y que poco a poco van corrigiendo este exceso mandando flujo del nodo hacia el sumidero o regresándolo hacia la fuente.

La idea de estos algoritmos es reducir el número de pasos de aumento de flujo a través de tres cosas:

1. Aumentando cada vez los flujos en grandes cantidades
2. Usando estrategias de combinatoria que limiten las trayectorias a considerar
3. Relajando la limitación de conservación de masa al menos en algunos pasos intermedios del algoritmo

Utilizando estas estrategias se puede limitar la complejidad computacional a $O(n^2\sqrt{m})$. Utilizando las estructuras de datos correctas. La complejidad reportada de hecho corresponde al algoritmo descrito arriba, utilizando una fila de prioridad con etiquetas de distancia.

Los algoritmos de flujo máximo utilizan el teorema de "flujo máximo-costo mínimo" para probar que ofrecen soluciones correctas, este es un teorema importante en teoría de las gráficas, sin embargo una demostración es un tema fuera del alcance de este proyecto. La idea es que si se encuentra al conjunto de aristas que si se remueven la gráfica tiene una componente conexa más, de forma que las aristas tengan la menor capacidad total posible, se está resolviendo automáticamente el problema del flujo máximo y viceversa.

4.2.2. Algoritmos para el problema de flujo con costo mínimo

Brevemente el problema de flujo máximo es: *Si el flujo a través de las aristas una red tiene un costo unitario asociado y debemos enviar flujos a partir de un vértice (o varios) y hacia otro (o varios), ¿Cómo podemos satisfacer las condiciones de envío y recepción al mínimo costo posible?.*

Al igual que sucede para los algoritmos de flujo máximo hay dos enfoques generales para resolver los problemas de flujo de mínimo costo:

- Comenzar desde el principio con soluciones factibles (que cumplen todas las constricciones) y moverlas hacia la optimalidad, o
- Comenzar con soluciones que no cumplen las constricciones pero que son óptimas y migrarlas hacia un estado donde poco a poco aseguramos el cumplimiento de las limitantes del problema.

Las soluciones en general incluyen alguna instancia de encontrar distancias mínimas o flujos máximos [2]. Las complejidades computacionales en general dependen del número de nodos n, el número de aristas m, la capacidad máxima de las aristas U y el costo más grande C. Y van desde $O(nU)$ hasta $O(mlogU)$.

Una vez finalizada la discusión de algoritmos en gráficas, procederemos a presentar el problema abordado en esta tesis.

Capítulo 5

Análisis del impacto de la arquitectura de red en el flujo vehicular

5.1. Descripción del problema

El objetivo de este trabajo de tesis es analizar el efecto de diferentes arquitecturas de red sobre la congestión del flujo vehicular conforme aumenta la densidad de automóviles presentes en el sistema. El análisis fue realizado a través de simulaciones computacionales basadas en agentes, que representan automovilistas, circulando a través de diferentes arquitecturas de red, específicamente una red de tipo Manhattan y la misma red con algunas modificaciones inspiradas en arterias características de la Ciudad de México.

Las simulaciones ocurren sobre redes predefinidas para cada arquitectura para las cuales se especifica la posición geográfica de sus vértices, que representan esquinas no semaforizadas en la ciudad. Las aristas representan las calles por las que circulan los automovilistas por lo que son aristas dirigidas o flechas y tienen propiedades que incluyen su capacidad y la velocidad máxima de circulación que un auto puede tener cuando viaja sobre ellas.

La simulación ocurre en pasos discretos, cada automóvil tiene un tiempo de salida definido así como información de cuál es su nodo inicial y su nodo final. Los automóviles que no han salido de su nodo inicial o que ya han alcanzado su nodo destino no se toman en cuenta para calcular la saturación de automóviles sobre las calles. Es por ello que para lograr simular diversas densidades de automóviles en el sistema se especifica que más autos tienen su tiempo de salida dentro de un mismo intervalo, lo que causa que más autos estén sobre las aristas al mismo tiempo.

Dado que estamos tratando con un sistema complejo, para lograr resultados significativos, una simulación con la misma arquitectura y los mismo automóviles (agentes con exactamente las mismas propiedades) es repetida en múltiples ocasiones y además son repetidos diversos escenarios de la misma arquitectura de red con la misma densi-

dad de automóviles pero no exactamente los mismos agentes. En cada simulación con los mismos agentes añadimos a la memoria de dichos agentes información sobre los días pasados, así que de hecho podemos tener una idea no sólo del comportamiento del sistema en condiciones preestablecidas, sino de la evolución del sistema a causa del cambio de rutas de los agentes a través de los días.

Haciendo uso de los resultados de las simulaciones reportaremos parámetros como la velocidad y tiempo promedio de recorrido de los conductores para cada densidad de automóviles presentes en la simulación y para cada arquitectura de red. También reportaremos en qué escenarios la simulación llega a un punto en el cual la red se encuentra totalmente congestionada y ya no hay ningún movimiento posible para los conductores.

5.2. Representación de los automóviles o automovilistas

En la modelación por agentes, un sistema está modelado como una colección de entidades autónomas y que tienen capacidad de tomar decisiones llamadas **agentes**. Cada agente tiene la capacidad de considerar su situación individual y tomar decisiones dentro de una serie de reglas predefinidas. En el nivel más simple un modelo basado en agentes consiste en una colección de agentes y la descripción de las relaciones entre ellos. Algo interesante es que aún en los modelos basados en agentes más simples, es posible encontrar patrones de comportamiento complejos y proveer información sobre la dinámica del sistema que modela [**?**].

Adicionalmente a las reglas fijas, es posible incorporar formas de memoria o aprendizaje en los agentes que permite que su comportamiento evolucione a través de la simulación.

Consideramos que esta forma de modelación se acopla con nuestros objetivos y por ello es el enfoque elegido en nuestras simulaciones.

Los automóviles, que son los agentes de nuestro trabajo, serán individuos guiados por una optimización egoísta, lo que quiere decir que su toma de decisiones se basa fundamentalmente en minimizar su propio tiempo estimado entre su origen y su destino a través del algoritmo A*. Para esto requerimos de especificar precisamente su nodo origen y su nodo destino, así como su estrategia de ruteo individual. Como las simulaciones pensadas en este trabajo ocurren durante el tiempo, también será necesario especificar el momento en que los conductores salen de su origen. Otra cosa importante es que en cada punto de la simulación los automóviles conozcan su posición en relación a los nodos y geográfica y la velocidad a la que van, esta velocidad debe ser registrada con el fin de simular el conocimiento adquirido sobre la red. basados en este conocimiento es que las rutas elegidas por cada agente se irán modificando a través de las repeticiones de la simulación.

La estructura que utilizaremos para representar a un automovilista o auto moviéndose en la red, a la que llamaremos `auto`, tiene los siguientes campos, que son inicializados al principio de la simulación y modificados a lo largo de ella:

- **o**: El vértice de la digráfica D donde el auto comienza su recorrido. Es a partir de este punto donde el auto aparece en la red una vez que es su tiempo de salida (descrito en breve).
- **d**: El vértice donde el auto finaliza su recorrido, en otras palabras, su destino. Una vez que el auto alcance este punto, este auto desaparecerá de la red y ya no ocupará espacio en ninguna de las aristas.
- **ts**: Un número real que indica el tiempo de salida (ts) del auto, todas las simulaciones comenzarán en un tiempo 0, subsecuentemente los automovilistas irán comenzando sus trayectos según su tiempo de salida. Antes de que ocurra este tiempo de salida un auto no ocupa ningún espacio físico de la red y por ello no satura la capacidad de ninguna arista de la misma.
- **k**: Constante k, que indica el perfil de riesgo del conductor, es decir cuánto confía en su propia memoria de días pasados y cuando usa un estimado basado en las velocidades máximas de recorrido de cada arista. El uso de esta constante quedará más claro en la sección 6.1
- **days_of_memory**: El número de días de memoria que cada conductor tendrá presente cuando calcule nuevas trayectorias.
- **speed_memories**: Un arreglo de diccionarios (forma de representar mapeos) que asocian índices de vértices de D con una velocidades. El hecho de que sea un arreglo, permite que se guarden las velocidades de varios días. La información de este arreglo es la que se usará para calcular nuevas trayectorias de mínima distancia en cada día de la simulación.
- **speed_memory**: Un diccionario que asocia índices de vértices de D con velocidades. Este es un único diccionario, que cada día de la simulación se reiniciará para registrar nuevos valores.
- **astarpath**: La trayectoria dada por el algoritmo A* entre el origen y el destino del nodo, el primer día se calcula con los tiempos de recorrido en cada arista suponiendo que es se viaja a la velocidad máxima permitida y se recorre la distancia euclidiana entre dos nodos, en días subsecuentes se usan diferentes heurísticas que serán descritas en la sección 6.1.
- **last_node**: El índice del último nodo visitado por el auto, este es un valor que se irá modificando conforme el auto recorra la red. Se inicializa con el valor del campo o.
- **next_node**: El índice del siguiente nodo que el auto visitará según su trayectoria, se inicializa como el destino de la primera arista del camino A* inicial.
- **avance**: Un valor de tipo real que representa la distancia que ha recorrido sobre la arista u, v, donde u es el último nodo por el que pasó y v es el siguiente nodo

según su trayectoria A*. Se inicializa en 0. y se actualiza cada vez que el auto avanza. Cada que el auto llega a una nueva arista se vuelve a resetear el valor en 0.

- **vel**: Un número real que representa la velocidad en un momento dado de la simulación, se inicializa en 0.
- **is_out**: Un valor de tipo booleano, que indica si el auto ya ha dejado su origen y comenzado su recorrido a través de la red, se inicializa como False.
- **llego**: Un número real que representa el tiempo, desde el tiempo inicial de la simulación completa (=0) donde el auto alcanza su nodo destino. Si se resta este valor del tiempo de salida ts se obtiene el tiempo que le tomó al conductor alcanzar su destino desde el momento que partió de su origen.

5.3. Representación de las redes

Las redes de nuestro trabajo pueden ser consideradas gráficas dirigidas, se desea que entonces se puedan extraer los vértices y las aristas que pertenecen a la gráfica y se puedan conocer las relaciones entre las componentes. Como la red estará representando una porción de una ciudad, los nodos y las aristas deben estar embebidos en un espacio geográfico, es decir que debe saberse la posición de los nodos y las longitudes de las aristas. También debemos tener información de la velocidad máxima registrada para cada arista, su capacidad máxima y el estado en cierto punto del tiempo, ya que parte de las suposiciones principales del trabajo es que la saturación de las aristas impacta la dinámica sobre las mismas. Tomando esto en cuenta también será importante conocer cómo exactamente la saturación impacta la velocidad o el tiempo de recorrido sobre cada arista.

La estructura que representa a una red está definida a través de tres componentes:

- **digraph**: Una digráfica $D = (V, A)$, específicamente del tipo `SimpleDiGraph`, definido en la paquetería `Graphs.jl` [15], este tipo representa una gráfica dirigida simple, lo que significa que no permite tener múltiples aristas entre un par de vértices.
- **position_array**: Un arreglo de posiciones de los vértices: Un vector de elementos en $\mathbb{R}^2$: $v = [(v_{1,1}, v_{1,2}), (v_{2,1}, v_{2,2}), ..., (v_{K,1}, v_{K,2})]$. Donde K es el número de vértices de la digráfica D, de hecho las $k = (1, 2, .., K)$ son una numeración de dichos vértices, la pareja $(v_{k,1}, v_{k,2})|k = (1, 2, .., K)$ representa las coordenadas en un plano cartesiano donde se ubica el k-ésimo vértice. Pensando en que la red de hecho está embebida en un espacio geográfico, que para las dimensiones de una ciudad bien puede ser pensado como una porción del plano real.
- **city_matrix**: Un arreglo tridimensional en Julia que contiene una serie de matrices de adyacencia de D, pensando que cada una es una propiedad de la calle a

la que la arista representa. De esta forma tendremos la representación de cuatro propiedades de cada calle:

1. El tiempo que harían al recorrer dicha calle si viajaran a la velocidad máxima.
2. El número de autos que caben en esa calle, para fines prácticos lo que haremos será multiplicar el número de carriles que deseamos por la longitud de la calle (distancia euclidiana entre las posiciones de sus puntas), expresada en metros y después dividir entre 5 metros, que representa la longitud promedio de los automóviles.
3. El número de autos que están en la calle en un momento dado del tiempo, después se explicará cómo se va modificando este número, pero para fines de inicializar la estructura de datos, todos los elementos de esta matriz comienzan siendo cero.
4. El tiempo que efectivamente tomaría a un conductor recorrer la calle en un momento dado del tiempo, este tiempo se calcula a través de la ecuación BPR (*Bureau of Public Roads*), descrita en la sección 2.6. Según una corrección del tiempo yendo a velocidad máxima que considera estado de saturación de cada calle (número de autos/capacidad).

5.4. Arquitecturas de red

Como se menciona en la descripción del problema, en este trabajo se analizaron redes tipo Manhattan así como redes tipo Manhattan con la inclusión de modificaciones simulando arterias importantes de la Ciudad de México, a continuación se describirá con más detalle cuáles fueron estas arquitecturas de red:

1. Red cuadrada sin diagonales

   Con esto nos referimos a una red tipo Manhattan que representa una porción de la ciudad donde las calles forman una cuadrícula y en todas las equinas los autos deben detenerse (o disminuir su velocidad) antes de cruzar. Las esquinas de esta red no se encuentran semaforizadas.

   Para representar lo antes descrito usaremos una red D con nodos representando las esquinas y las aristas representando calles con la característica de ser todas de la misma longitud. Cada nodo de la gráfica subyacente de D tiene a lo más cuatro vecinos. Para la digráfica se especifica el sentido de las calles y en la red tipo Manhattan se trabaja con sentidos alternados, de forma tal que todas las aristas que tienen la misma longitud y latitud tienen la misma dirección y estas direcciones se van alternando entre hileras y filas.

   Es importante mencionar que cuando se realizaron pruebas sobre esta red se descubrió que para casos impares de número de nodos por lado, la red no era fuertemente conexa (no era posible alcanzar cualquier nodo destino desde cualquier

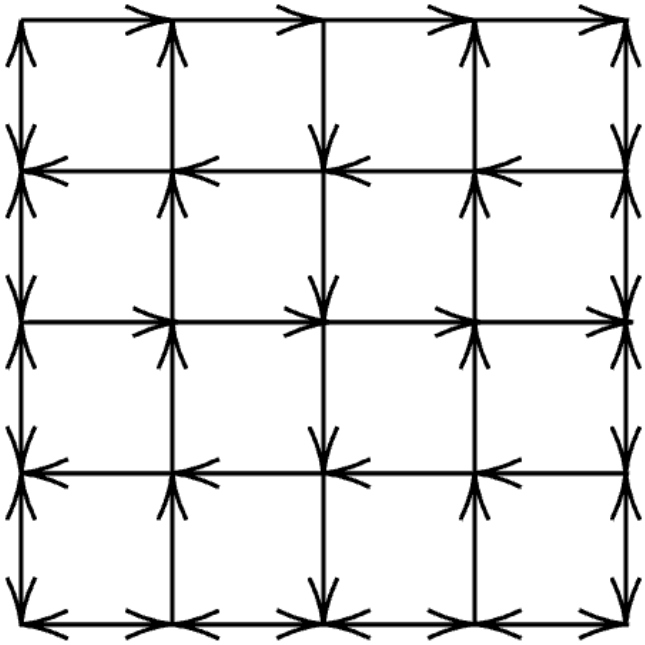

Figura 5.1: Red cuadrada de 5 nodos por lado con la corrección mencionada en el texto para asegurar conexidad fuerte.

nodo origen), para corregir este problema se decidió agregar el sentido inverso de las calles en tres de las orillas de la red. Como se muestra en la imagen 5.1.

La situación que sucede en las esquinas de la ciudad, que es que los automovilistas deben frenar antes de seguir su camino cuando llegan a una intersección fue representada a través de incorporar un nodo y una arista adicional, de una longitud que permita sólo un auto a la vez, entre cada par de vértices $u, v \in V | (u, v) \in A$ para la digráfica de la ciudad $D = (V, A)$, justo antes de v, dado que v es el exvecino de u. Esta subdivisión geográfica, representada en la imagen 5.2, se realiza para cada arista dirigida de la digráfica D.

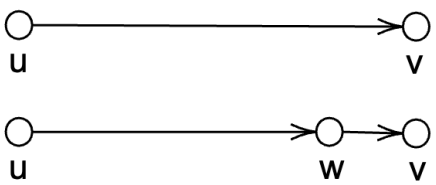

Figura 5.2: Proceso de subdivisión de aristas para reducir la capacidad antes de las esquinas de la ciudad, provocando una disminución de la velocidad en las esquinas.

2. Red tipo Churubusco

   La segunda arquitectura de red que definimos fue a la que llamamos tipo Churubusco, Avenida Río Churubusco es una arteria de la Ciudad de México que tiene la característica de estar compuesta mayoritariamente por puentes y túneles, por lo que al viajar sobre ella, los conductores no requieren de detenerse a causa de intersecciones con otras calles.

   Para nosotros la red tipo Churubusco es una red Manhattan como la descrita en la sección anterior, que incluye esquinas lentas y que es modificada con la adición de una diagonal, a la cual los autos pueden acceder a través de las aristas que forman la cuadrícula. La diagonal cumple la característica de que, una vez que se transita sobre ella, no es necesario disminuir la velocidad en las intersecciones.

   A diferencia de Río Churubusco, en nuestra representación la diagonal tiene un solo sentido. Esta arquitectura se muestra en la imagen 5.3.

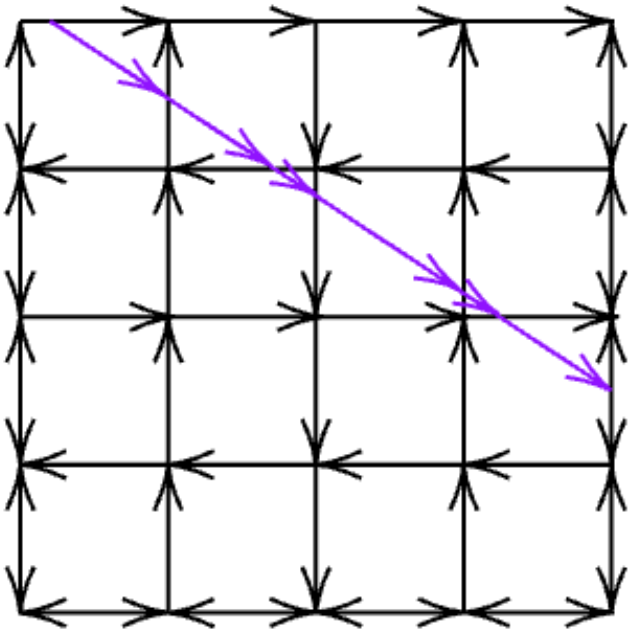

Figura 5.3: Representación de la arquitectura de red tipo Churubusco, la diagonal se resalta en morado, indicando la dirección de la diagonal.

3. Red tipo División del Norte

   La tercer arquitectura de red considerada está inspirada en Avenida División del Norte, otra arteria de la Ciudad de México, esta avenida es una diagonal que atraviesa otras vialidades y que no está compuesta por puentes y túneles, por lo que en muchas esquinas tiene semáforos.

En nuestra representación no añadiremos la semaforización, sin embargo en cada intersección de la diagonal, forzaremos una disminución de la velocidad de los conductores a través de la subdivisión de las aristas que componen a la misma, algo que no se realizó para la arquitectura tipo Churubusco. Esta subdivisión sólo se realizó sobre las aristas que tenían cierta longitud mínima (al menos 15m) porque de lo contrario corríamos el riesgo de que existieran aristas con una capacidad de menos de un automóvil. La diagonal tiene un solo sentido.

Esta tercer arquitectura se puede observar en la imagen 5.4, en la que se resaltan las subdivisiones de las aristas en color rojo, en esta imagen se puede notar que no todas las aristas de la diagonal se encuentran subdivididas, lo que es debido a la situación explicada en el párrafo anterior.

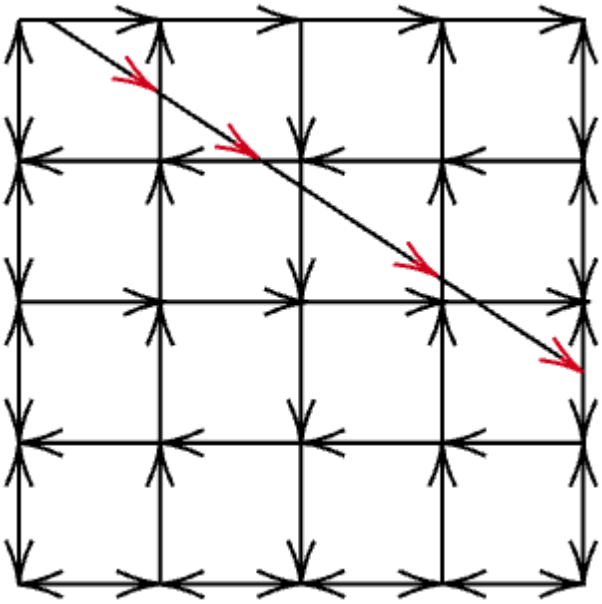

Figura 5.4: Representación de la arquitectura de red tipo División del norte, las subdivisiones de aristas sobre la diagonal se resalta en rojo, indicando la dirección de la diagonal.

5.4.1. Adición de diagonales

Para el lector interesado en cómo se añadieron diagonales a las redes cuadradas, se explican los pasos de este procedimiento a continuación:

1. Definir las coordenadas donde comenzará la diagonal, la ordenada al origen (donde interceptaría la diagonal si continuara hasta $x = 0$ en el plano cartesiano) y la pendiente.

2. Dadas las propiedades anteriores y sabiendo cuántos vértices hay por lado de

la cuadrícula de la red y la longitud de las aristas, Calcular las intersecciones que la diagonal tendría con la cuadrícula y guardar las posiciones de dichas intersecciones. Las descripciones utilizando ordenada al origen y pendiente que se implementan en el trabajo, fueron seleccionadas de forma tal que ninguna intersección ocurre en la posición exacta de un vértice $v \in V$ que pertenezca a la digráfica $D = (V, A)$ que representa a la red.

3. Por cada una de las intersecciones, crear un nuevo vértice $v \in V$ de la digráfica $D = (V, A)$ que representa la red. Llamemos $W \subset V$ al conjunto de vértices que son agregados en este paso.

4. Por construcción, cada vértice agregado $w \in W$, yace geográficamente sobre una arista $(u, v) \in A$. El último paso será, subdividir cada arista $(u, v) \in A$ tal que w se encuentra sobre (u, v). La subdivisión se define como:

   - Crear una arista (u, w)
   - Crear una arista (w, v)
   - Eliminar la arista (u, v)

Las diagonales están pensadas para correr en un sólo sentido, peor es posible hacer modificaciones para considerar casos más generales.

En el capítulo siguiente se dan detalles de cómo se llevaron a cabo las simulaciones en las tres arquitecturas de red presentadas, para diferentes densidades de automóviles.

Capítulo 6

Detalles de la implementación computacional

Todo el código presentado en este trabajo está disponible en un repositorio público de GitHub, accesible en: https://github.com/LuzMarianaBlaz/Code-MarianaTesis.

6.1. Algoritmos de menor distancia

El algoritmo que se propone en este trabajo para emular la búsqueda del camino más corto de un punto a otro de la ciudad por parte de los conductores, es una modificación al algoritmo A* del que se discutió con anterioridad, en la sección 4.1.2.3.

Como se describió en dicha sección la idea de A* es evaluar un camino posible que pase por un nodo n, a partir de dos componentes: $g(n)$ que es el costo de alcanzar n y $h(n)$ que es el costo de alcanzar el destino a partir de n. En total tendremos entonces $f(n) = g(n) + h(n)$, que puede ser interpretada como "el costo estimado de la solución menos costosa que pasa por n". Una vez que se determina la forma en que se estima el costo de pasar por cierto nodo, el procedimiento consiste en ir construyendo un camino del origen al destino deseado a partir de utilizar los nodos posibles de menor costo. La determinación de qué heurística es la que se usará para estimar costos es lo que al final determina cómo será la estrategia.

En general las heurísticas que se utilizaron en este trabajo tienen que ver con la velocidad que los automovilistas "consideran"que llevarán cuando atraviesen los nodos en la red, en la descripción de cada heurística se explicará cómo concluyen esto los agentes. Una vez que el automovilista tiene la noción de cuál será la velocidad en un nodo $v \in V$, a la que llamaremos $vel(v)$ y cuál será la velocidad en el nodo destino de su trayecto $t \in V$ $(vel(t))$, la heurística $h(v)$ se calcula como: $h(v) = \frac{|t-v|}{(vel(v)+vel(t))/2}$. Donde $|t-v|$ es la distancia euclidiana entre v y t y $(vel(v)+vel(t))/2$ es un promedio simple de las velocidades estimadas en v y en t. Nótese que a pesar de que no se está

haciendo un promedio a través de los posibles trayectos entre v y t, esto es funcional ya que una vez que A* elija avanzar al nodo $n \in V$, dado que n es el nodo de menor costo, a partir de dicho n recalculará cuál es el nodo siguiente de menor costo para alcanzar t, por lo que en realidad se está haciendo una optimización en cada paso.

A su vez el costo de alcanzar un nodo $n \in V$ a partir de otro $w \in V$, descrito como $g(n)$, se calcula como: $g(n) = \frac{|n-w|}{(vel(v)+vel(t))/2}$ lo que es directamente la longitud de la arista $(w, n) \in A$ dividido por un promedio simple de las velocidades de su cola y su punta.

A continuación se explica cómo es que cada agente determina cómo es $vel(n)$ para un nodo $v \in V$

Nota: Los agente, u objetos tipo auto en realidad guardan las velocidades cada vez que se alcanza un nuevo nodo, y la velocidad queda registrada como asociada al nodo que es la cola de la arista que dejan.

1. A* individual

   En este caso nuestra $vel(n)$ es una combinación entre la memoria de un automovilista (el registro que el agente tiene de la velocidad de las aristas que ya conoce) y una parte estimada que tiene que ver con el tiempo que tardaría en recorrer una arista si es que pudiese transitar a través de ella con la velocidad máxima permitida.

   La heurística final es una combinación de dos heurísticas, la que el conductor obtiene de las velocidades que recuerda y la que estima para las velocidades que no conoce, asumiendo que podrá desplazarse a la velocidad máxima, entonces: $h_{final}(n) = (1-k)*h_{estimada}(n)+k*h_{memoria}$, donde $h_{estimada}$ es la estimación de tiempo que tardaría dado que circula a la velocidad máxima permitida, $h_{memoria}$ es el tiempo asociado a la velocidad que el agente registró que llevaba cuando pasó por ese vértice y k es una constante que puede ser determinada para cada conductor según su perfil de riesgo.

   Tanto $h_{memoria}$ como $h_{estimada}$ son calculadas tal y como se describió antes de comenzar esta enumeración, utilizando la distancia euclidiana y un promedio simple de velocidades.

2. A* con memoria colectiva

   Esta es una extensión de el A* individual, ya que la heurística en este caso también toma en cuenta una parte estimada y una parte de memoria individual, sin embargo, se agrega una componente adicional que representa la memoria de todos los agentes en colectivo. Esto se decidió incluir para tomar en cuenta la información que se pasa "de boca en boca" o el conocimiento general que los conductores de una ciudad tienen sobre la misma. La heurística aquí será: $h(n) = (1-k) * h_{estimada} + k * (0.8 * h_{mp} + 0.2 * h_{mc})$, donde $h_{estimada}$ es la estimación de tiempo (euclideano) que tardaría dado que circula a la velocidad máxima permitida, h_{mp} (memoria propia) es el tiempo asociado a la velocidad que el agente registró que llevaba cuando pasó por ese vértice, h_{mc} (memoria

colectiva) es un promedio de la memoria propia de todos los automovilistas y k es una constante que puede ser determinada para cada conductor según su perfil de riesgo.

3. A* para conductores omniscientes

   La heurística para esta versión consiste en eliminar la parte estimada y de memoria individual y utilizar como heurística únicamente un conocimiento colectivo del estado de la red que se va diluyendo con el paso de los días como $\frac{1}{2^n}$ donde n es el número de días que han pasado. Específicamente en este caso:

$$h^n(v) = \frac{|t - v|}{(vel^{(}_{colectiva} n - 1)(v) + vel^n_{colectiva}(t))/2}$$

   donde $vel^n_{colectiva}(v) = \frac{vel^n_{colectiva}(v) + vel^n_{promedio}}{2}$, siendo $vel^n_{promedio}$ el promedio de las velocidades de todos los autos en el día $n - 1$.

6.1.1. Implementación computacional de las heurísticas de menor distancia

La forma que se utilizaron estas heurísticas y los algoritmos A* en forma práctica, fue utilizando la función `a_star` del paquete de julia `LightGraphs.jl` [29]. Definiendo como parámetros de entrada:

- **g**: La gráfica dirigida de entrada, que en nuestro caso es la representación de red $D = (V, A)$ de una porción de la ciudad. Esta representación es la misma para todos los conductores.
- **s**: El vértice inicial $s \in V$ del trayecto del agente correspondiente.
- **t**: El vértice final $t \in V$ del trayecto del agente correspondiente.
- **distmx**: La matriz de pesos que será utilizada para determinar el costo de alcanzar un nodo $n \in V$, en el trayecto, esta matriz es la que da la componente $g(n)$ de la heurística para A*. En este caso, dado que consideramos que para cada conductor se habla de una distribución de tiempos distinta, *distmx* variará de agente en agente, y estará determinada por las mismas componentes que cada heurística. Es decir en el caso de A* con memoria individual, la matriz de pesos será una combinación del tiempo estimado (con las velocidades máximas permitidas) y el tiempo según la propia experiencia del conductor (a partir de las velocidades que registró al pasar por cada nodo). En el caso de A* con memoria colectiva será lo mismo más el tiempo según la experiencia colectiva. Y en el caso de A* para conductores omniscientes será el conocimiento colectivo del estado de la red.

- **heuristic**: la heurística también es diferente para cada agente y corresponde al caso que estemos trabajando. En el caso de A* con memoria individual es la combinación de tiempos estimados y tiempos recordados, calculado como la distancia euclidiana dividido entre la velocidad promedio entre el nodo $n \in V$ y el nodo final $t \in V$. Para A* con memoria colectiva es lo mismo pero los tiempos estimados, propios y colectivo se calculan como la distancia euclidiana dividida entre la velocidad máxima, la velocidad promedio entre el nodo $n \in V$ y el nodo final$t \in V$ que cada conductor registró, y un análogo con la velocidad colectiva, respectivamente. En el caso de A* para conductores omniscientes será simplemente el tiempo calculado del conocimiento colectivo del estado de la red.

Para cada conductor (agente) el camino óptimo dado por el algoritmo es utilizado durante todo un "día"de simulación hasta que logra llegar a su destino o bien la red llega a un estado donde no hay más movimientos posibles. Una vez que esto ocurre, cada conductor puede recalcular su camino óptimo.

6.2. Descripción de "un día de simulación"

Para un conjunto de objetos tipo automóvil y un objeto red, cada "día de simulación"se entiende como la corrida completa de un algoritmo en el que todos los automovilistas logran completar su ruta A* desde su nodo origen y hasta su nodo destino o bien la red llega a un estado en el que ya no hay movimientos posibles.

El día de simulación se compone de pasos discretos que se evalúan dentro de un ciclo `while` (una estructura de control en la programación que se utiliza para repetir un bloque de código mientras se cumpla una condición específica) que precisamente se detiene cuando todos los autos llegaron a su destino (evaluado con la condición `auto.llego!=0.`) y dentro del cual se implementa una cláusula de tipo `break` (o de interrupción) en el caso donde no hay más movimientos posibles en la red.

Los pasos discretos corresponden a acciones que hacen los agentes dentro de la red y pueden ser de dos tipos: Un agente sale de su nodo origen ó un agente alcanza un nuevo nodo en la red. En cada paso nos aseguramos de que todos los autos evolucionen su estado según el delta de tiempo (Δt) que toma la acción, es decir que todos los autos que se encuentran recorriendo la red avanzarán una distancia $\Delta t \times vel$ sobre la arista que recorren. El hecho de dividir el tiempo en estos pasos discretos nos asegura que durante la simulación nunca sucederá que un conductor avance en exceso sobre una arista de la red, es decir que vaya viajando sobre una arista uv, alcance el nodo v y siga considerando que va sobre la arista uv, o bien que "se pase"de una esquina sin considerar que ya cambió de calle.

El ciclo se inicializa con dos arreglos vacíos, uno para contener tiempos y otro para contener autos atorados, y una matriz de adyacencia para la red con todas las entradas en ceros. Para comenzar el día de simulación, tres parámetros son requeridos: un valor llamado `tiempo_universal`, un conjunto de objetos tipo auto ya inicializados y un

objeto tipo red. La variable de `tiempo_universal` lleva la cuenta de cuánto tiempo ha pasado sumando el delta de tiempo de todos los pasos discretos que ya han ocurrido, y con esto no se hace referencia al tiempo que toma la simulación sino al tiempo que toma a un conductor completar el paso discreto (alcanzar la esquina o salir de su nodo origen), según su velocidad reportada. Llevar la cuenta de este `tiempo_universal` es útil para poder saber el tiempo de recorrido de los automovilistas (tiempo de llegada - tiempo de salida) y para saber cuándo un nuevo auto debe salir de su nodo origen (acción que sucede cuando `auto.ts` = `tiempo_universal`)

El fragmento de código que se repite dentro del ciclo consiste en realizar los siguientes pasos en orden:

1. Se calcula cuál es el siguiente auto en salir y cuál es su tiempo de salida. Es decir se identifica al objeto `auto` con la propiedad `ts` menor. Por la construcción de todos los conjuntos de autos de este trabajo, siempre se asegura que nunca dos autos tendrán el mismo tiempo de salida.

2. Se calcula cuál es el siguiente auto en alcanzar un nuevo nodo (cambiar de arista) y en cuánto tiempo esto sucederá.

3. Se registra el delta de tiempo en el que ocurrirá la próxima acción, es decir se calcula el mínimo entre el delta de tiempo antes del siguiente cambio de arista y el delta de tiempo antes de la siguiente salida de un auto de su nodo origen. Si no hay ninguna próxima acción posible, se detiene la simulación.

4. Si no se detuvo la simulación se le suma el delta de tiempo al tiempo universal, y este a su vez se guardo en el arreo tiempos a través de una operación `push!`. Posteriormente, según qué acción sigue suceden una de dos cosas:

   - Si sigue un tiempo de salida se marca la propiedad del auto que indica que ya salió de su origen como verdadera y se actualizan su nodo anterior u y su nodo siguiente v, después se aumenta en uno al contador del número de autos de la arista uv, es decir que se suma uno a la entrada u, v de la matriz en `Red.city_matrix` que lleva la cuenta de cuántos autos hay en cada arista.
   - Si sigue un cambio de arista se guarda la velocidad del auto en la entrada correspondiente al nodo cola de la arista que el auto está dejando. Es decir que si iba en una arista (u, v) y llega al nodo v, la velocidad queda registrada en el nodo u. Después se resta en uno el contador de autos en la arista (u, v) y se revisa v es el nodo destino del automovilista. Cuando los autos llegan a su destino no se hace ningún cambio posterior sobre la cuenta de autos en las aristas, el único cambio que se realiza es sobre la propiedad del auto que indica en que momento llegó a su destino, registrando en este valor el `tiempo_universal` de la simulación. Si con el cambio el auto no alcanza su nodo destino todavía, se averigua cuál es la siguiente arista que debe visitar según su camino A* para ese día, digamos la arista (v, w), se registra v como el último nodo visitado y w como su próximo nodo por visitar,

posteriormente se suma uno al contador de cuántos autos están en la arista (v, w)

5. Independientemente de cuál haya sido la acción que marcó el avance discreto del tiempo universal, es importante que actualicemos a todos los automóviles considerando el avance que tendrán según su velocidad. En eso consiste el siguiente paso a través de iterar sobre todos los autos (excepto sobre el auto que realizó la última acción discreta) y aumentar su distancia sobre la arista en la que se encuentran en su velocidad multiplicada por el delta de tiempo de la última acción. En este paso no es necesario actualizar el contador de los autos que están en las aristas, ya que por cómo calculamos cuál sería el siguiente paso sabemos que en principio ninguno de los otros autos debería realizar un cambio de arista. La velocidad que se usa en esta actualización no es la última registrada en un nodo, sino una que se calcula para cada auto dentro del cálculo del próximo cambio de arista como: longitud por recorrer entre la arista dividida entre el tiempo que le tomaría recorrerla según la función BPR (**??**), propiedad que se guarda en una de las componentes de `Red.city_matrix`.

6. Como último paso de cada repetición del ciclo precisamente se actualizan los tiempos de recorrido en cada arista según la corrección BPR tomando en cuenta qué tan llenas están las aristas (cuál es la proporción de autos en cada arista entre su capacidad máxima). Y se guarda esta corrección en `Red.city_matrix`.

Una consideración importante de la implementación de las rutinas de búsqueda del siguiente tiempo de salida y el siguiente cambio de arista es que sólo consideran movimientos posibles, es decir movimientos que implican que el auto se moverá hacia una arista *donde todavía hay espacio*, lo que quiere decir que el número de autos en dicha arista es menor a su capacidad máxima, según el estado de la entrada `city_matrix` de la red.

Para los autos que no pueden cambiar de arista, no podemos seguir la regla exacta descrita en el paso 5 de la lista anterior. Porque es posible que el siguiente cambio de arista viable (hacia una arista que aún tiene espacio) ocurra en un Δt mayor al de una arista no viable. Entonces para evitar que estos autos "se pasen"de su siguiente nodo, lo que hacemos es indicar que están "a punto"de alcanzarlo, marcando su distancia como la longitud de la arista sobre la que van menos una distancia aleatoria y diminuta.

Para los autos que no logran salir en su tiempo de salida especificado, lo que hacemos es agregar una pequeña cantidad de tiempo adicional para que intente su salida en un momento posterior.

Cuando aún hay autos en la red y sin embargo ningún auto más puede salir y ninguno puede realizar un cambio de arista cuando se entra a la cláusula `break`, el código se suspende y decimos que "la red se saturó".

Cuando el ciclo termina, ya sea porque todos los autos alcanzaron su destino o porque la red se saturó, se pueden obtener las siguientes propiedades de la simulación: El arreglo de tiempos discretos, la velocidad promedio de en la red durante toda la

simulación, la densidad final de la red (cuántos autos y en qué aristas quedaron) y un arreglo que guarda el número de autos que había en la red en cada paso y si en dicho paso la red se atoró, este último extremadamente útil para analizar las densidades en las que se satura la red según diferentes arquitecturas. Ademas de esta información, cada auto conserva la información de sus propiedades, cuyos valores se fueron actualizando durante todo el día de la simulación, por ejemplo las velocidades que registró durante todo su trayecto. En el objeto `Red` también queda el registro el último estado de la propiedad `city_matrix` dentro de la simulación.

6.3. Extracción de la información de un día de simulación

A partir de la misma función de día de simulación es posible algunos datos que nos permitirán obtener conclusiones de cómo sucede el movimiento de los autos en la red, específicamente podemos obtener la velocidad promedio de la red y la distribución de autos que quedaron en el caso de que la red se atasque y los números de autos en los que ocurrió que la red se saturara (si es el caso). Otra información importante puede ser extraída a través de consultar el estado final de cada auto. Específicamente, después de cada día de simulación se obtiene:

- La distancia recorrida por cada auto, a través de sumar las longitudes de las aristas de su camino A*.
- El tiempo de recorrido de cada auto, este tiempo se obtiene restando su tiempo de llegada menos su tiempo de salida, para autos que nunca llegaron y que por tanto esta diferencia es negativa, se coloca un valor `NaN` que en este caso representa que no se tiene información.
- La velocidad promedio de viaje de cada auto, obtenida de dividir su distancia recorrida entre el tiempo promedio.

6.4. Reinicialización de la red y los autos

Cuando un día de simulación termina, la red y los autos contienen la información del último paso de la simulación. La red contendrá entre otras cosas la información de las últimas velocidades reales según la función BPR, y, en el caso donde se haya atascado, la información de cuántos autos hay en cada arista (cuando no se atora también tiene esta información, pero esta es idéntica al caso inicial, es decir una matriz de ceros). Los autos contendrán la información de que todos están en su destino, el tiempo en el que lograron llegar, etcétera.

Una vez que guardamos la información del día de simulación, para poder comenzar exitosamente una nueva iteración, debemos reinicializar a cada automovilista y a la red. En la mayoría de los parámetros esto quiere decir regresar los valores de cada propiedad

a su estado inicial (de cuando el objeto fue creado), pero no en todos los casos es así, por ejemplo, para poder conservar la propiedad de que los autos deben recalcular sus rutas según su propia memoria, no tendría sentido reinicializar este valor por completo. A continuación se da una lista de los pasos que suceden en el paso de reinicialización:

Primero se realizan las siguientes operaciones en orden a todos los objetos de tipo `auto`:

1. Se restablecen los tiempos de salida originales, utilizando una copia del mismo que se guarda al comenzar la simulación. Este paso es necesario porque en el momento donde un auto quiere salir pero no puede por que las aristas están saturadas, se agrega una pequeña cantidad de tiempo para que intente salir en una iteración posterior.
2. Se marca el avance sobre su arista `auto.avance` y su velocidad `auto.vel` como 0
3. Se marca la propiedad `auto.is_out` como falsa y el tiempo en el que llega (`auto.llego`) como 0.
4. Se indica que el último nodo `auto.last_node` es el nodo origen.
5. Se guarda la memoria del día que ya terminó en `auto.speed_memories` y se reinicializa una memoria vacía para el día que va a comenzar.
6. Se calcula un nuevo camino A* según los recuerdos del auto y se indica si hay un cambio respecto al A* anterior, en caso de que así sea, se registra el índice del auto para el que esto ocurre.
7. Se guarda como nodo siguiente (`auto.last_node`) el nodo que sigue al origen según el camino A* recién calculado.
8. Se coloca la posición del auto como la posición del nodo inicial (si se están haciendo animaciones, en otro caso este paso no es importante).

Posteriormente se restablecen el número de autos en la red, marcando este valor como igual a cero en todas las aristas y con eso se calcula la matriz de velocidad en la red con la función BPR, que al no haber autos en la red, será igual a la de velocidades máximas reportadas.

El propósito de la función es realizar cambios sobre los objetos, pero ya que se estaba calculando, se decidió que esta función devolviera el arreglo de índices de los autos que en esta operación recalcularon su camino A* a uno diferente.

Cabe aclarar que el guardado del resumen de la memoria (`auto.speed_memories`) sucede de forma distinta según el escenario en el que estemos:

- Si los automóviles sólo consideran su propia memoria para tomar decisiones, es decir cuando se usa A* individual: Simplemente se recorren los índices del arreglo de memoria (que tiene un tamaño fijo **para cada auto**), de forma que se remueve la entrada más vieja cuando ya se ha alcanzado el tamaño del arreglo, para dar espacio a la memoria del día que justo finalizó.

- Si los automóviles consideran una parte de memoria individual y otra con memoria colectiva, es decir que se usa A* con memoria colectiva: Se hace el mismo procedimiento que el indicado en el inciso anterior, pero además hay un arreglo adicional que contiene los últimos 40 días de memoria colectiva (promedios de las memorias individual de cada repetición), en este arreglo también se van recorriendo los elementos de forma de que cuando se excede el tamaño se elimina la entrada más antigua para dar espacio a un nuevo registro.

- Si los automóviles tienen la información completa de todas las velocidades, es decir que se usa A* para conductores omniscientes: En este caso también se modifican las memorias de cada automóvil y la memoria colectiva. En este caso tanto los arreglos individuales como el arreglo de memoria colectiva tienen sólo dos entradas, lo que puede resultar extraño, sin embargo, se eligió esta implementación porque justamente permite que a través de calcular el promedio de dos entradas (la más reciente y una que contiene la información de todos los días pasados) obtener la información en el formato que se deseaba para el algoritmo de menor distancia, es decir que considere una dilución de la memoria $\frac{1}{2^n}$ donde n es el número de días que han pasado. La implementación es muy sencilla: Si el arreglo tiene ya dos entradas $a = [e_1, e_2]$ y se tiene una nueva información m_i, lo que se hace es calcular el promedio entre e_i y e_2 y registrar esta información en la segunda entrada de a (sustituyendo a e_1), por su parte la nueva información m_i sustituye a e_1 en la primer entrada del arreglo.

6.5. Una simulación a través de varios días

Ya se ha explicado en secciones anteriores cómo sucede un día de simulación, como se extra información de dicho día y cómo se reinicializan los objetos para poder comenzar una nueva repetición, en esta sección se menciona como se incorporan estas componentes para tener varios días de simulación, con conductores que tienen orígenes y destinos iguales durante todos los días, pero que van aprendiendo sobre las condiciones de la red a través de los días y que por tanto, van intentando viajar por nuevas rutas.

Una simulación de varios días, se entiende como la iteración en varias ocasiones de los pasos sucesivos: simular un día, extraer valores y reinicializar. Para realizar esto se utiliza un ciclo `while`, en el cual se iteran estos tres pasos en múltiples ocasiones y en el cual, la condición que se evalúa cada vez es si ya se cumplió un número de días especificado.

Antes de comenzar el ciclo, se deben especificar:

- **Qué red vamos a utilizar**: Se construye la red con el número de nodos por lado, una condición que indica si las calles de la gráfica base serán de doble sentido o no, indicando si tendrá o no una diagonal, de qué tipo será esta diagonal, donde comienza y su pendiente. Además de cuáles son las longitudes de arista, sus velocidades y su capacidad.

- **Qué agentes vamos a usar**: Es decir cuántos autos lanzaremos en la red, cuáles serán sus tiempos de salida y llegada, cuál es su distribución de parámetro h (de perfil de riesgo)
- Por **cuántos días** vamos a correr la simulación completa, que es el valor que se utiliza para detener el ciclo while.

Una vez que se tiene dicha información se genera el número de autos deseado, de forma que la distribución de orígenes y destinos sea homogénea en toda la red, que la distancia que deben recorrer sea mayor que un número específico y que además los tiempos en los que salen también sean en intervalos regulares y sin repetir valores entre los autos. También se construye la red deseada (sin diagonal, con diagonal tipo Churubusco o con diagonal tipo División del Norte), siguiendo el procedimiento descrito en la sección 5.4 con el número de vértices por lado, de doble sentido o no, ciertas velocidades, capacidades y distancias, según las especificaciones.

Ya que se tiene la red a utilizar, los agentes que viajarán a través de ella y el número de veces que debemos simular, guardar y reinicializar, se realiza el ciclo while, en orden lo que sucede es que:

1. Se simula un día con la función de un día de simulación, la red y los autos.
2. Se guardan las distancias, tiempos y velocidades de los autos, así como la velocidad promedio en la red, el estado final de esta y en qué número de autos sucedió que se atoró la red.
3. Se reinicializan los autos y la red y se guardan los índices de autos que cambiaron su camino A*.
4. Se guarda la información de velocidades, tiempos, índices, velocidades en la red, matriz de densidad final y en cuántos autos se saturó la red.

El guardado de la información se realiza utilizando archivos de tipo `jld`, que es una forma de archivo que se usa en el lenguaje `Julia` y que permite recuperar objetos tal y como son definidos en el código, es decir que no es necesario transformar arreglos, diccionarios o incluso objetos definidos por el usuario en estructuras tabulares para guardarlos [30].

6.6. Diferentes escenarios considerados y sus parámetros

En este trabajo se corrieron simulaciones utilizando diferentes arquitecturas de red, número de automóviles y tipo de memoria, los parámetros en común para todas las simulaciones se pueden encontrar en la tabla 6.1.

Utilizando siempre los mismos parámetros, las variaciones que hicimos fueron:

1. Red sin diagonal, con conductores omniscientes

2. Red con diagonal tipo Churubusco, con conductores omniscientes
3. Red con diagonal tipo División del Norte, con conductores omniscientes
4. Red sin diagonal, con memoria colectiva
5. Red con diagonal tipo Churubusco, con memoria colectiva
6. Red con diagonal tipo División del Norte, con memoria colectiva

Las variaciones se realizaron en la construcción de la red en la simulación de varios días, reiniciando las memorias según el caso y seleccionando la heurística correspondiente para ser utilizada en el algoritmo A*.

Tabla 6.1: Parámetros comunes en todas las simulaciones

Parámetro	Valor
Número de autos	De 600 a 1700 en pasos de 50
Distribución del parámetro h	0.5 para todos los autos
Periodo de tiempo en el que se lanzan los autos	De 0 a 150 segundos
Distancia mínima de los trayectos	5 aristas
Número de vértices por lado de la red	5
Doble sentido	No
Pendiente de la diagonal	$-\pi/5$, si aplica
Inicio de la diagonal	$[7., 160.]$, si aplica
Longitud de las calles	40 m
Distribución de velocidades	Homogéneo, 12 m/s
Número de días simulados	100

Para poder extraer información de las diferentes arquitecturas independientemente de las distribuciones de orígenes y destinos, cada caso discutido se repitió en 10 ocasiones. Hay que considerar que la distribución de orígenes y destinos también es aleatoria y homogénea a través de la red. Por eso consideramos que con este número de iteraciones es suficiente para entender el tipo de regímenes de tiempos y velocidades que llegan a suceder en diferentes arquitecturas de red y tipos de memorias.

Los parámetros que informan la construcción de la diagonal, forman una estructura como la de la figura 6.1, la diferencia entre la red con diagonal tipo Churubusco y la diagonal tipo División del Norte es que sobre las calles de la diagonal tipo División del Norte hay una subdivisión en una arista que hace que los conductores tengan que "frenar en las equinas", situación que no sucede en la diagonal tipo División del Norte.

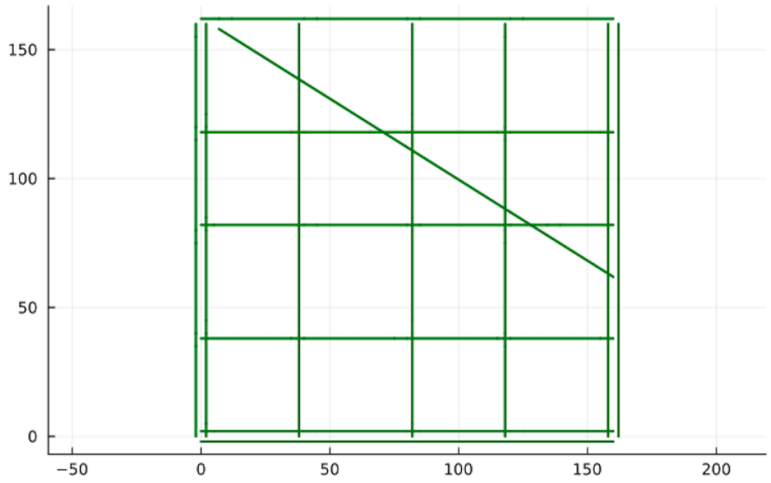

Figura 6.1: Arquitectura de red según los parámetros especificados para la diagonal.

6.7. Extracción de la información a partir de las simulaciones de varios días

En la sección 6.3 se explica cómo se extrae la información a partir de la simulación de un día, después en la sección 6.5 se menciona que esta información se va guardando como parte de las iteraciones que representan diferentes días continuos. En esta sección se explicará cómo se procesa esa información guardada para extraer condiciones generales de lo que sucede en cada arquitectura, con cada tipo de memoria considerada, a diferentes densidades de automóviles y a través de las 10 iteraciones mencionadas en la sección 6.6.

Recordemos que la información se queda guardada en archivos tipo `jld`. Estos archivos guardan, para cada día de simulación, la siguiente información:

- Las distancias de recorrido de todos los autos.
- Los tiempos de recorrido de todos los autos.
- Los índices de los autos que cambiaron su camino A* de ese día al siguiente.
- Las velocidades promedio en las aristas de la red.
- La matriz de densidad
- Un arreglo para cada día que indica para cada paso en la simulación, el número de autos que había en la red y si esta se atoró o no en ese punto.

en ese orden específico.

Para poder extraer información sobre la dinámica de los autos en general, según el tipo de red y la memoria, se procesa la información obteniendo valores promedio por día en cada simulación con el fin de entender la evolución de las dinámicas a través de los días. También se hacen agregaciones de todas las simulaciones para el mismo tipo de memoria y arquitectura de red en diferente número de autos, para entender cómo cambian los sistemas según la densidad. Este preprocesamiento se hace después de correr todas las simulaciones para un mismo tipo de memoria y mismo tipo de arquitectura de red, siguiendo los siguientes pasos:

1. Se obtienen la media y la moda de la distancia, velocidad, tiempo e índices de cambios de camino A* de los autos por día de simulación completa, agregando la información de todos los autos. Este paso se realizó asignando a los valores nulos el mínimo de los valores que sí están disponibles para las velocidades o el máximo de los valores disponibles para los tiempos. Estas correcciones no son necesarias para la distancia ya que las distancias planeadas por día sí quedan registradas independientemente de que el auto alcance su destino.

2. Se obtienen la media y la moda de la distancia, velocidad y tiempo por auto, promediando a través de todos los días. Aquí se usan las mismas variaciones descritas en el paso anterior.

3. Se obtiene información por día acerca del estado final de saturación de la red y de la velocidad.

4. Se condensa la información de las relaciones entre el número de autos y el estado de saturación.

La forma en que se obtiene la información condensada de las relaciones entre el número de autos y el estado de la simulación es a través de tres operaciones, la información de cada día está dada como $[[n_1, c_1], [n_2, c_2], ...[n_k, c_k]]$, donde n_i es el número de autos en el paso i del día y c_i es la condición de si la red se atascó o no a ese número de autos, donde $c_k = 0$ si no se atascó y $c_k = 1$ si sí se atascó.

La información de condensada se utiliza para generar gráficas acerca de la simulación específica con un número de autos y repetición de la simulación completa a través del número de días. Posteriormente se guardan (en el orden especificado):

- El promedio de velocidades en las aristas de la red a partir del día 20.

- El promedio de saturaciones no cero (el promedio del estado de las aristas al final de los casos donde la red se atascó), o bien una matriz conteniendo ceros, si nunca ocurrió que la red se saturase.

- El promedio de la velocidad, el tiempo y la distancia de todos los autos y considerando todos los días a partir del número 20, para tener datos más estables.

- Un diccionario de autos atorados, que indica en las llaves el número de auto y en los valores un arreglo con cuántas veces ocurrió ese número de autos en la red y la proporción de esas veces que sucedió una red atascada.
- La proporción de días atorados del total de 100 veces de la simulación a través de varios días.

Para este punto tendremos un archivo con esta información resumida por tipo de memoria, arquitectura de red, número de autos e índice de la repetición (del 1 al 10).

Esta información se vuelve a agregar para obtener información por tipo de memoria, arquitectura de red y número de autos, lo que queremos son valores iguales a los descritos en el listado anterior, pero agregados a través de las iteraciones. En esencia esto se consigue sacando promedios.

Es con estos últimos valores que se obtienen los resultados presentados en el trabajo, la implementación de las funciones puede ser consultada en el repositorio del código utilizado en la tesis.

6.8. Intento (fallido) de cálculo del flujo de costo mínimo y del precio de la anarquía

En la sección 4.2 se habló de algunos algoritmos para encontrar los flujos máximos o flujos de costo mínimo (problema descrito en 1.2), para este trabajo se intentó calcular el tiempo promedio de los automóviles en las mismas redes que se utilizaron, con el fin de calcular un Precio de la Anarquía con este tiempo proveniente del flujo de costo mínimo como denominador del cociente. El intento no fue exitoso por las siguientes razones:

La primera es que no es posible simular un día completo de tráfico en la forma que lo hacemos aquí, permitiendo que los autos esperen antes de avanzar si no hay capacidad en la arista que deben utilizar, además de que en general las implementaciones buscan un óptimo en estado estacionario. En general el óptimo en estado estacionario podría contener a lo más la suma de las capacidades de las aristas de la red, es decir aproximadamente 450 autos en un mismo momento.

La segunda es que, aún si buscamos enviar sólo los autos máximos permitidos por la red, en cada nodo sólo puede llegar un número de autos a lo más la suma de las capacidades las aristas incidentes a los nodos, si la demanda o la generación de autos de un nodo es más de este valor, siempre tendremos problemas que son marcados desde un principio como inviables.

La tercera es que aún si consideráramos sólo problemas viables, es decir que la demanda o la generación de autos de un nodo fuera de a lo más la capacidad de sus aristas incidentes (unos 16 autos aproximadamente sin esquinas lentas y tan solo 2 con esquinas lentas), estos algoritmos normalmente no consideran el tiempo de espera de los

conductores que están al final de la fila de la arista, y que además el tiempo promedio se aumenta en forma exponencial con el flujo de la arista.

Por estas razones, las soluciones que se lograsen encontrar con el algoritmo estarían muy alejadas de los escenarios simulados en este trabajo y no es posible encontrar un cálculo del precio de la anarquía a través de ellas.

Aunque no hayamos tenido éxito, a continuación se reporta la forma en que realizamos las pruebas:

El cálculo de flujo de costo mínimo se realizó con la función `mincost_flow`, integrada en la paquetería `LightGraphsFlows` de `Julia` [3]. Esta función requiere de una digráfica, un vector con demandas de los nodos, una matriz de adyacencia con las capacidades de las aristas, otra matriz con el costo e las aristas, un optimizador y, como argumentos opcionales una demanda de aristas, un nodo fuente o un nodo sumidero (estos dos últimos sólo se usan si no se especifican los vectores de demandas).

El optimizador puede ser cualquier solver de problemas de Programación Lineal, en los intentos realizados en este trabajo se utiliza el paquete `Clp`, una interfaz del solver `COIN-OR Linear Programming` [18].

Se utilizaron las mismas redes, matrices de capacidad y matrices de costo (tiempos sin considerar la función BPR) que en las simulaciones de este trabajo y se determinaron aleatoriamente nodos fuente y sumideros a través de la gráfica con demandas menores (en valor absoluto) a las capacidades de sus aristas incidentes.

Reiteramos que al final se decidió no utilizar estos resultados puesto que no es posible comparar los escenarios con los que se simularon en este trabajo.

Capítulo 7

Resultados y análisis

7.1. Diferencias en la dinámica de los autos según la arquitectura de red

El resultado principal es una diferencia encontrada en la relación de los parámetros medidos del sistema:

- Puntos de saturación o atascamiento de la red
- Tiempos de recorrido promedio de los automóviles
- Distancias promedio de los automóviles
- Velocidades promedio de los automóviles
- Cambios de ruta de los automóviles

Con la densidad de los automóviles (número de automóviles lanzados en cada simulación), en las diferentes arquitecturas de red. Se harán comparaciones entre la red sin diagonales, la red con diagonal tipo Churubusco y la red con diagonal tipo División del Norte, según lo descrito en la sección 6.6. Toda la información es extraída según los métodos presentados en la sección 6.7

7.1.1. Proporción de veces en que se saturan diferentes arquitecturas de red según el número de autos

Las gráficas 7.1 y 7.2 representan la proporción del total de simulaciones (repeticiones × número de días) de los casos en los que la red se satura, para los casos de memoria colectiva y conductores omniscientes respectivamente.

Como se observa, **el punto donde comienzan a ocurrir saturaciones sucede en un menor número de autos cuando hay una diagonal** tipo División del

Norte, seguido de donde hay una diagonal tipo Churubusco, finalmente para la red sin diagonales la saturación de la red ocurre a un número mayor de autos. Consideramos que esto está relacionado con los cuellos de botella presentes en la red, es decir las aristas donde se debe reducir la velocidad, adicionalmente a un fenómeno de extensión a la paradoja de Braess 2.4. Le llamamos una extensión, debido a que, en nuestro caso también es cierto que *"A partir de un sistema donde la toma de decisiones de rutas es egoísta, una mejora en la red puede degradar el desempeño de la red."* pero, en el caso original de la paradoja se habla de ruteos donde todos los automóviles salen de un mismo punto y llegan a un mismo punto también y de flujos en estado estacionario, y en nuestro caso los orígenes y destinos están distribuidos en toda la red y las salidas de los automóviles ocurren en diferentes momentos de la simulación.

El comportamiento es parecido cuando se habla del punto donde la red se satura en todas las ocasiones. Para las redes con diagonal esto ocurre antes, posiblemente porque si se llega a saturar la diagonal, esencialmente se divide a la gráfica en dos componentes conexas y se eliminan las trayectorias posibles entre uno y otro lado de la red.

En las gráficas 7.1 y 7.2, se colocan unas líneas verticales en donde ocurre la transición entre una red no saturada y una red que se satura en todas las simulaciones enviadas. Estas líneas verticales también se colocarán en las siguientes secciones para mostrar qué pasa con los demás parámetros obtenidos (distancias, velocidades, tiempos, etc.) en esos puntos, que son de importancia porque pueden considerarse una transición de fase en la dinámica de la red.

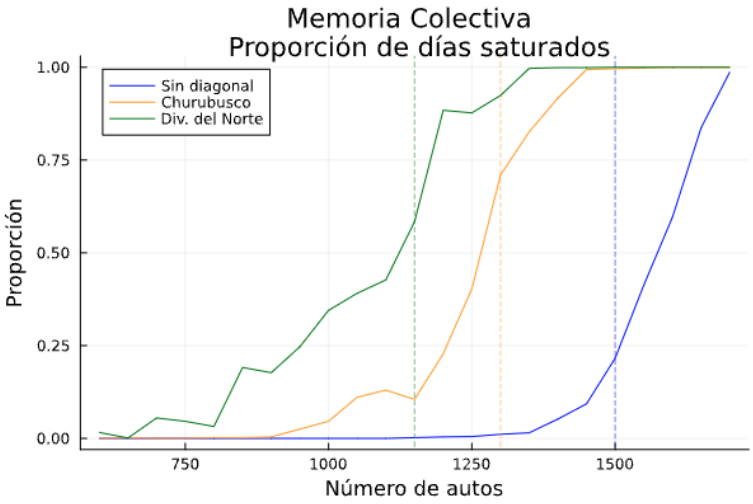

Figura 7.1: Proporción de simulaciones saturadas según la arquitectura y el número de autos; Simulaciones con memoria colectiva.

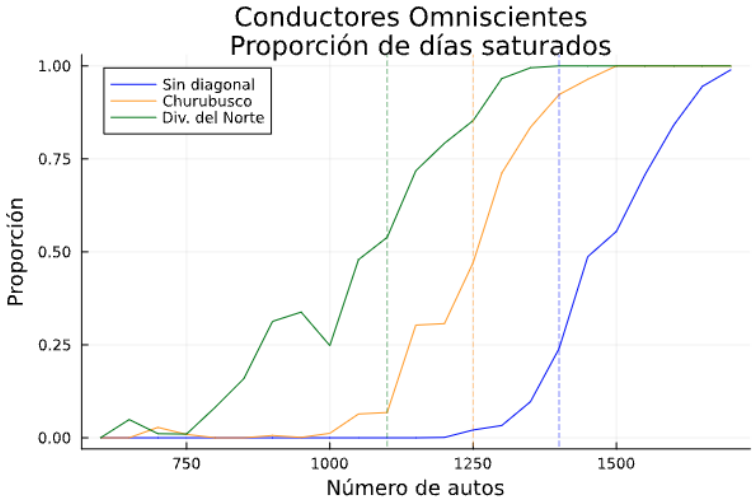

Figura 7.2: Proporción de simulaciones saturadas según la arquitectura y el número de autos; Simulaciones con memoria colectiva.

7.1.2. Lugares en los que se satura la red para las diferentes arquitecturas

Cuando se analiza la distribución de automóviles en el punto en el que la red se satura, es posible tener más información de dónde ocurren precisamente los cuellos de botella en cada caso:

- En la red sin diagonales, la saturación se alcanza en momentos donde las calles se saturan más o menos homogéneamente en toda la red 7.3. No hay regiones especiales que al congestionarse saturen a la red. Las saturaciones en la red cuadrada es que no ocurre con la congestión en un punto específico, sino un momento en que ocurran múltiples congestiones en un número importante de puntos en la red.
- En la red con diagonal tipo División del Norte, el estado de saturación se alcanza cuando se llena la diagonal ó cuando se llenan los puntos de acceso a la diagonal 7.4. A diferencia de el caso de la red sin diagonal, no sólo es una cuestión del número de congestiones, sino de su localización específica. Recordando que al saturar la diagonal estamos separando la gráfica en dos componentes conexas.
- En la red con diagonal tipo Churubusco, sucede algo similar a la red con diagonal tipo División del Norte, excepto porque la saturación de la diagonal sucede más tardíamente, no así con los puntos de acceso y salida de la misma 7.5.

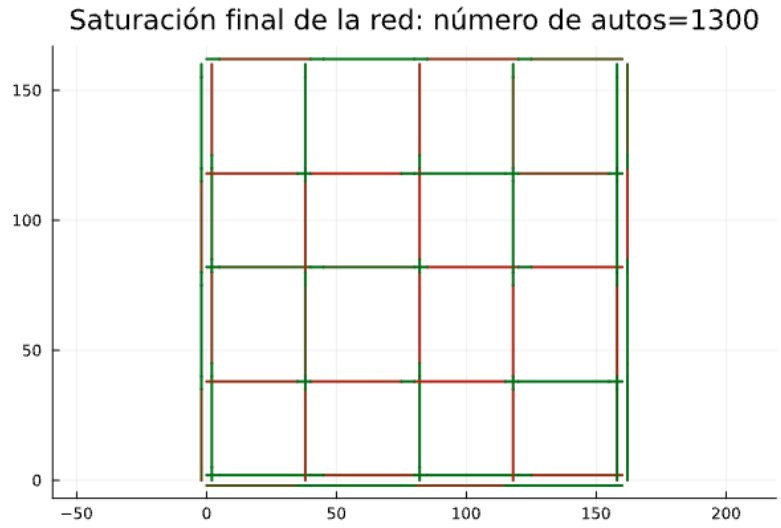

Figura 7.3: Estado final promedio de la saturación para una red cuadrada **sin diagonales** con 1300 autos lanzados para la simulación a través de varios días; El color rojo indica un estado de saturación mayor.

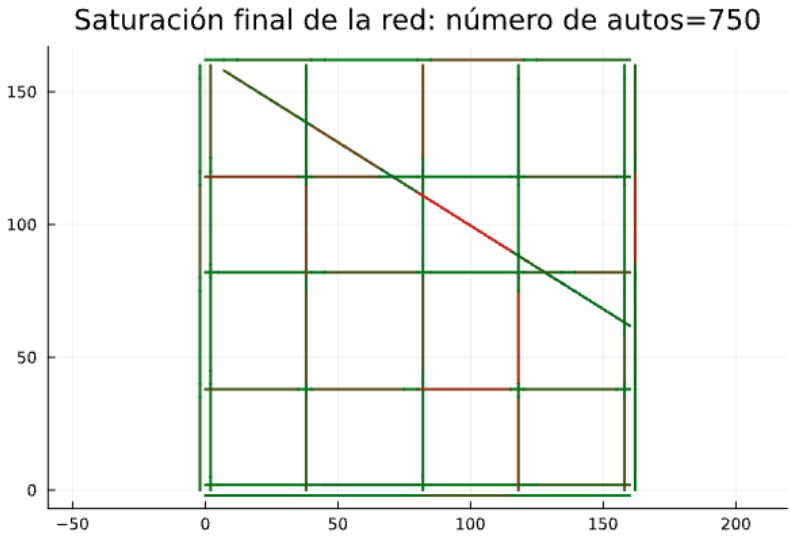

Figura 7.4: Estado final promedio de la saturación para una red cuadrada con diagonal tipo **División del Norte** con 750 autos lanzados para la simulación a través de varios días; El color rojo indica un estado de saturación mayor.

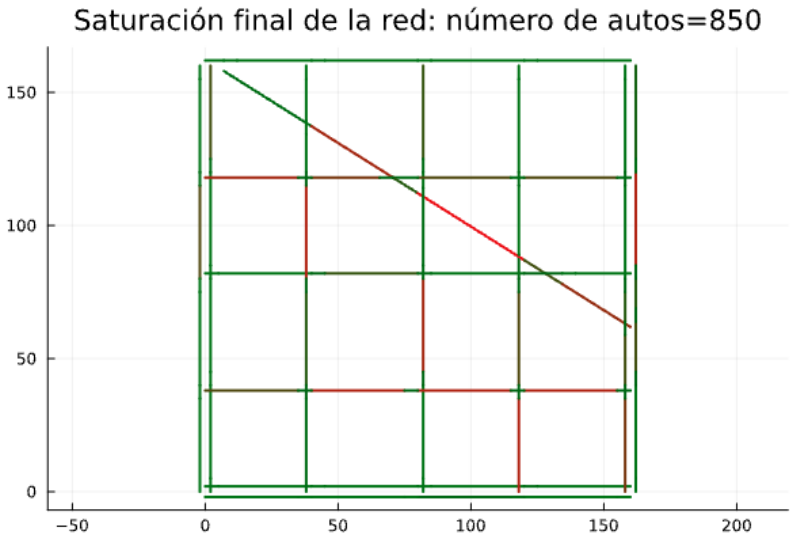

Figura 7.5: Estado final promedio de la saturación para una red cuadrada con diagonal tipo **Churubusco** con 850 autos lanzados para la simulación a través de varios días; El color rojo indica un estado de saturación mayor.

7.1.3. Tiempo de recorrido promedio por auto según la arquitectura de red

En las gráficas 7.6 y 7.7 podemos notar que en general el tiempo de recorrido de los autos que logran alcanzar su destino comienza en un valor menor para las redes que contienen diagonales y es prácticamente igual independientemente del tipo de diagonal.

Conforme aumenta la densidad de automóviles el tiempo incrementa rápidamente en el punto de transición del estado de saturación (líneas verticales), esto ocurre mucho antes para las redes con diagonal. Y, entre los dos tipos de diagonal, ocurre antes en la diagonal donde los conductores deben detenerse (tipo División del Norte).

A partir de densidades intermedias (en nuestras simulaciones unos 1,000 autos) la red sin diagonales presenta mejores tiempos de trayecto que las redes con diagonales, aún si la diagonal no requiere de que los conductores se detengan al viajar sobre ella.

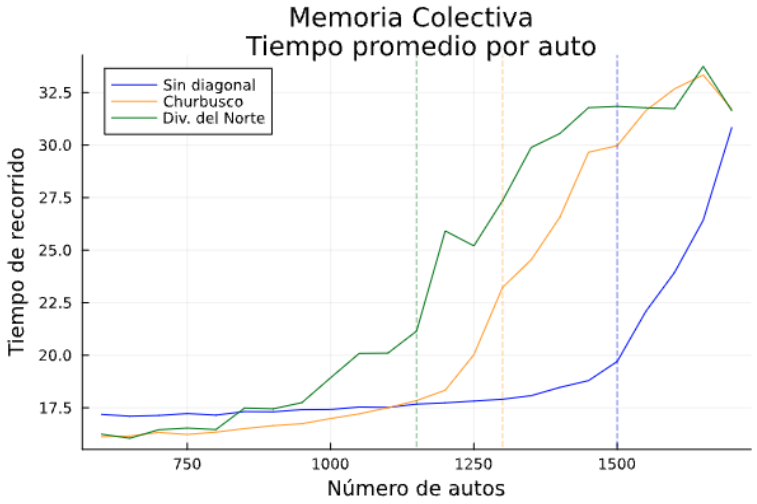

Figura 7.6: Tiempo promedio por auto según diferentes arquitecturas y número de autos presentes en la simulación; Memoria colectiva.

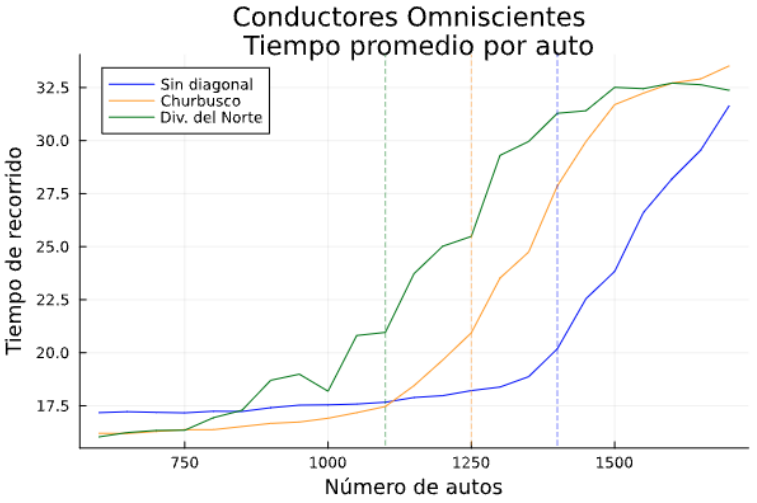

Figura 7.7: Tiempo promedio por auto según diferentes arquitecturas y número de autos presentes en la simulación; Conductores Omniscientes.

7.1.4. Distancia de recorrido promedio por auto según la arquitectura de red

En las gráficas 7.8 y 7.9 podemos notar que la distancia promedio es menor en el caso donde hay una diagonal presente, de cualquier tipo y que esta distancia es mayor

(un incremento relativo de 0.08) cuando no hay una diagonal.

Las distancias se mantienen estables a través de todas las densidades de automóviles, lo que quier decir que, a pesar de todo, los conductores siguen eligiendo aprovechar las diagonales.

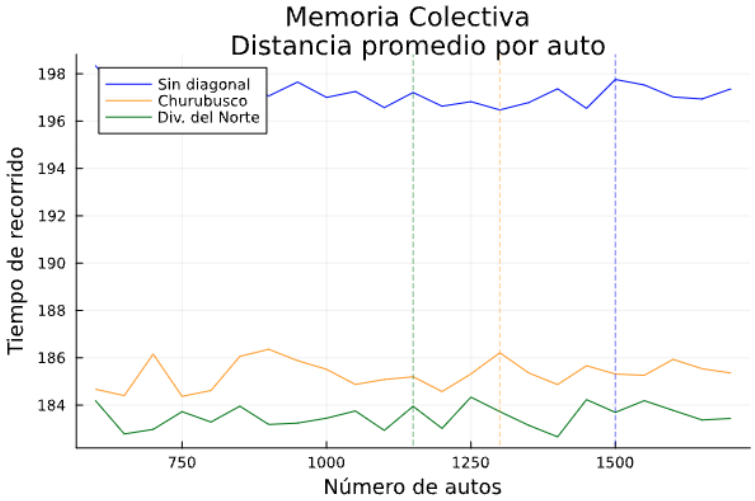

Figura 7.8: Distancia promedio por auto según diferentes arquitecturas y número de autos presentes en la simulación; Memoria colectiva.

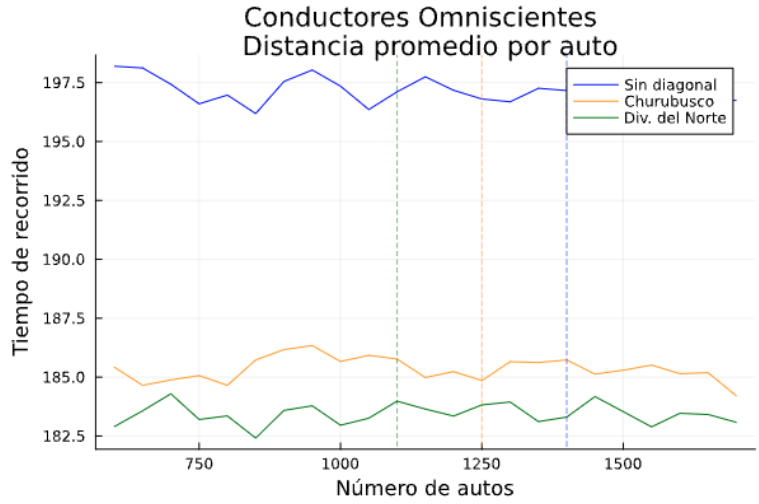

Figura 7.9: Distancia promedio por auto según diferentes arquitecturas y número de autos presentes en la simulación; Conductores Omniscientes.

7.1.5. Velocidad promedio en el recorrido por auto según la arquitectura de red

En las gráficas 7.10 y 7.11 podemos notar que la velocidad promedio es mayor en el caso donde la arquitectura de la red no contiene ninguna diagonal, seguido de el caso de la que contiene una diagonal tipo Churubusco. Las menores velocidades se reportan en la red con diagonal tipo División del Norte.

En todos los casos, todas las aristas tienen una velocidad máxima de 12 m/s, por lo que todos los cambios en la velocidad promedio se deben a estados de saturación de la red, y como ya hemos visto esta saturación ocurre primero en las redes que presentan algún tipo de diagonal.

Es interesante ver que en los puntos de transición de saturaciones (líneas verticales), se observan cambios de concavidad en el régimen en el cual disminuye la velocidad.

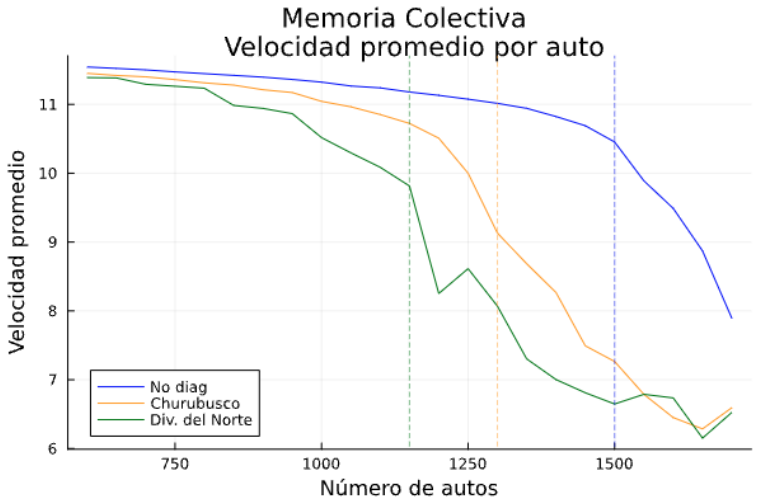

Figura 7.10: Velocidad promedio por auto según diferentes arquitecturas y número de autos presentes en la simulación; Memoria colectiva.

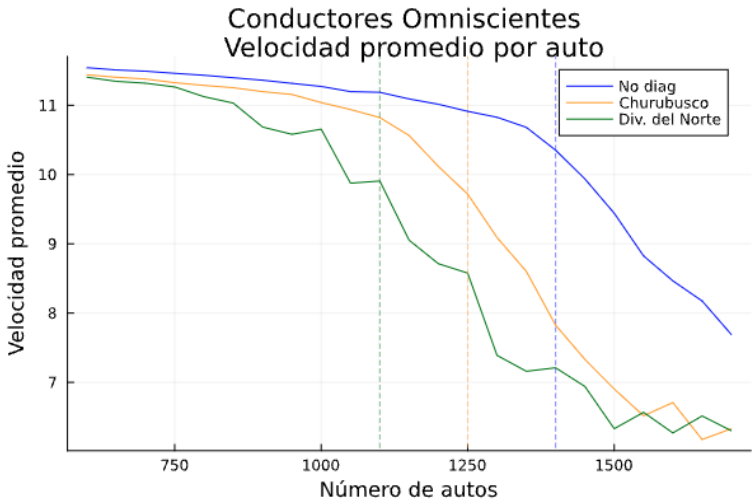

Figura 7.11: Velocidad promedio por auto según diferentes arquitecturas y número de autos presentes en la simulación; Conductores Omniscientes.

7.1.6. Autos cambiando de ruta según la arquitectura de red

En la gráfica 7.12 se observa que el número de cambios de ruta promedio por día crece conforme aumenta la densidad de autos, hasta que se alcanza el punto de transición en la saturación de la red, en este punto se alcanza el máximo del número de cambios y comienza a decaer, posiblemente por que los autos que nunca logran llegar a su destino no llegan a registrar la información de las aristas que nunca lograron alcanzar en la ruta del día k y por ende, no tienen información adicional que justifique un cambio de ruta en el día $k + 1$.

En la gráfica 7.13 no se observa que se alcance este máximo, la tendencia sigue creciendo conforme aumenta el número de autos lanzados a la red, en este caso la memoria de todos los autos es compartida, lo que quiere decir que incluso los autos que no logran alcanzar su destino se benefician del conocimiento compartido de los demás.

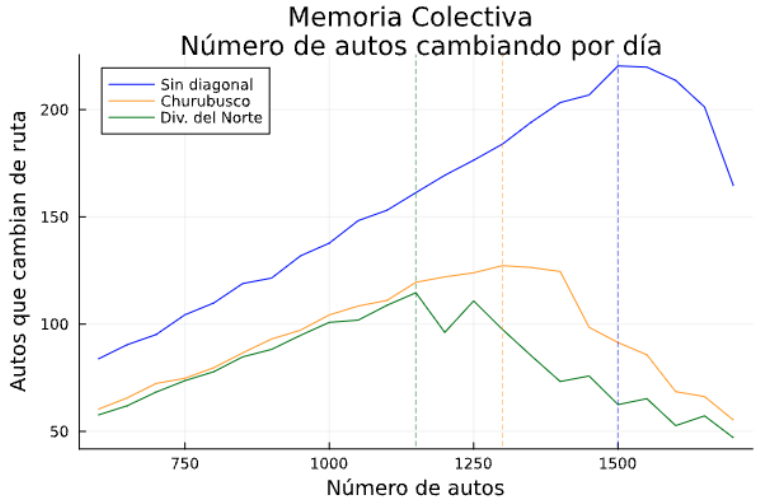

Figura 7.12: Número de autos cambiando de ruta según diferentes arquitecturas y número de autos presentes en la simulación; Memoria colectiva.

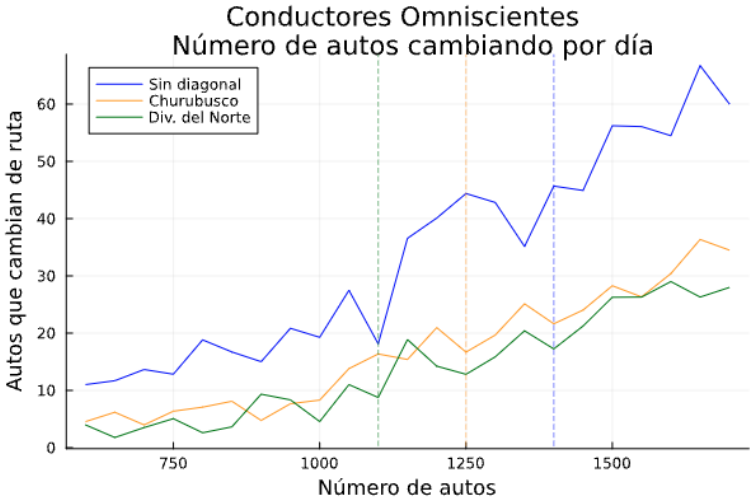

Figura 7.13: Número de autos cambiando de ruta según diferentes arquitecturas y número de autos presentes en la simulación; Conductores Omniscientes.

7.2. Resultados adicionales sobre las diferencias entre arquitecturas

En la sección 6.3 se explica que uno de los parámetros extraídos de las simulaciones es el número de autos que hubo en cada paso discreto y si en ese momento la red se saturó o no. En la sección 6.7 se explica cómo se condensa esa información para obtener resultados generales para una arquitectura de red y una densidad de automóviles. En escancie, de estos datos podemos obtener dos cosas:

1. La frecuencia con la que un número de autos estando en la red al mismo tiempo ocurre en las simulaciones.
2. La frecuencia con la que ese número de autos en la red al mismo tiempo provoca que la red se atasque.

Un resultado interesante es que las gráficas de estos dos parámetros son mucho más similares entre arquitecturas de red que las demás:

- En los tres casos hay una forma similar a una campana en la gráfica de frecuencia de aparición 7.14, a densidades bajas, la frecuencia de apariciones es baja, posiblemente porque, al haber pocos autos en la red una de dos cosas ocurre: Estos se dispersan rápidamente (ocupando pocos pasos discretos), o bien, la densidad crece rápidamente porque la red aún tiene capacidad. A densidades intermedias, el número de apariciones es muy grande, puesto que cada paso discreto representa apenas un cambio de arista y es posible que a esas grandes densidades los automóviles requieran muchos más pasos discretos antes de alcanzar su destino (reducir el número de automóviles en la red). Las densidades altas son difíciles de alcanzar porque es posible que la red se sature antes de que se alcance ese número de autos, por lo que esos valores aparecerán pocas veces.
- En los tres casos, la frecuencia de saturaciones ocurre más o menos al mismo número de autos **en la red simultáneamente** 7.15, un número entre 300 y 400 autos en la red al mismo tiempo (la capacidad máxima de la red en cualquier momento, es decir la suma de las capacidades de las aristas es de 416 autos para la red sin diagonal y 450 para las redes con diagonal). Esto quiere decir que la diferencia entre los estados de saturación de la red según el tipo de arquitectura, se deben al número máximo de autos que se encuentran en la red simultáneamente se alcanza en números distintos de autos que se lanzan en la simulación 7.16. El hecho de que un número de autos que satura a la red se alcance a menores autos lanzados en la simulación para redes con diagonal, se debe a que estos tardan más en llegar por la competencia por la diagonal, y antes de que logren llegar a su destino más autos son incluidos en la red.

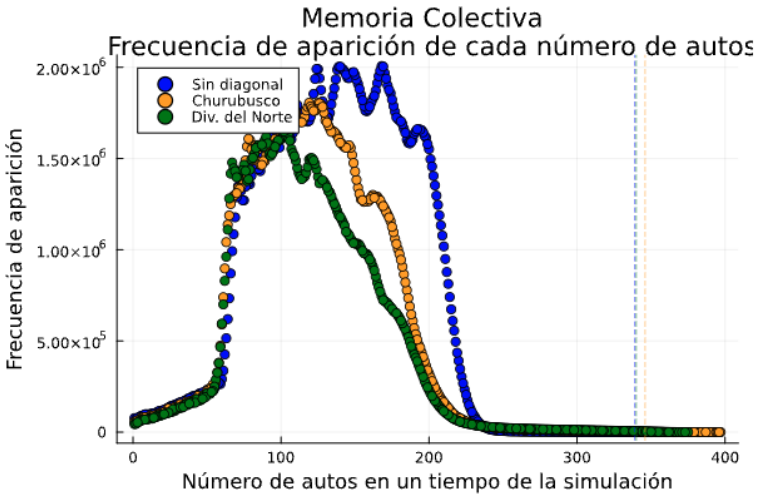

Figura 7.14: Frecuencia de aparición de cada número de autos al mismo tiempo en la red; Memoria colectiva.

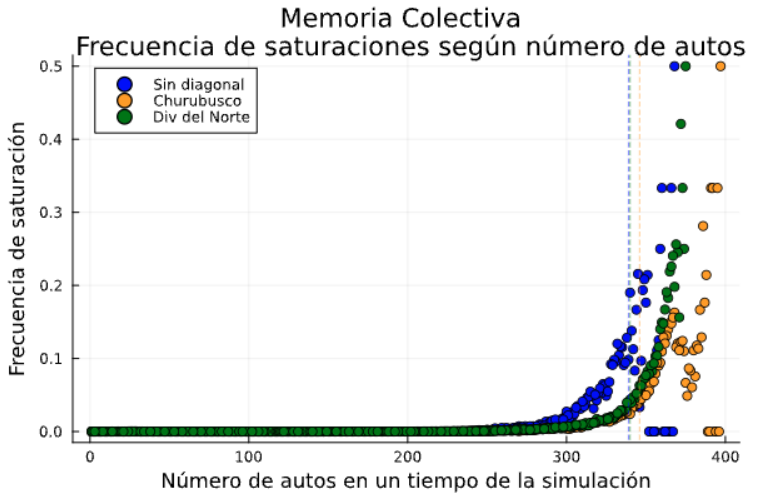

Figura 7.15: Frecuencia de saturación de cada número de autos al mismo tiempo en la red; Memoria colectiva.

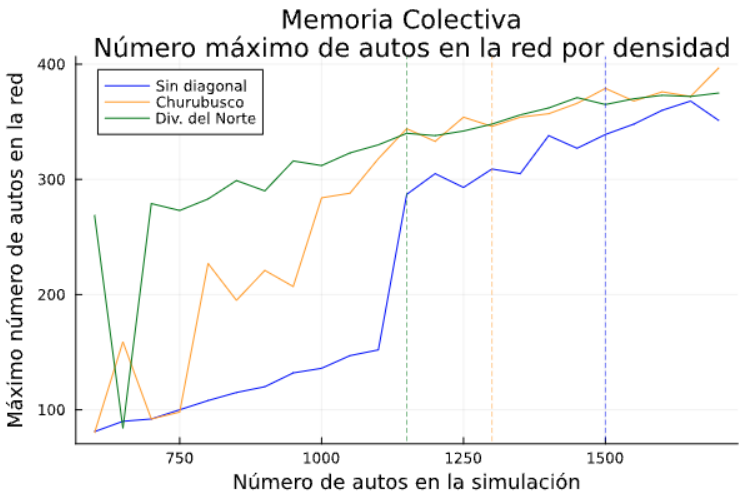

Figura 7.16: Número de autos en la red al mismo tiempo según el número de autos lanzado en la simulación.

7.3. Conductores omniscientes contra conductores con memoria colectiva

La diferencia entre las simulaciones con memoria colectiva y memoria de tipo omnisciente es que las de memoria colectiva consideran una parte individual y otra colectiva para cada conductor, mientras que las de tipo omnisciente utilizan sólo una parte colectiva fuertemente cargada hacia los eventos más recientes 6.1.

Figura 7.17: Saturación según el tipo de memoria; Red sin diagonal.

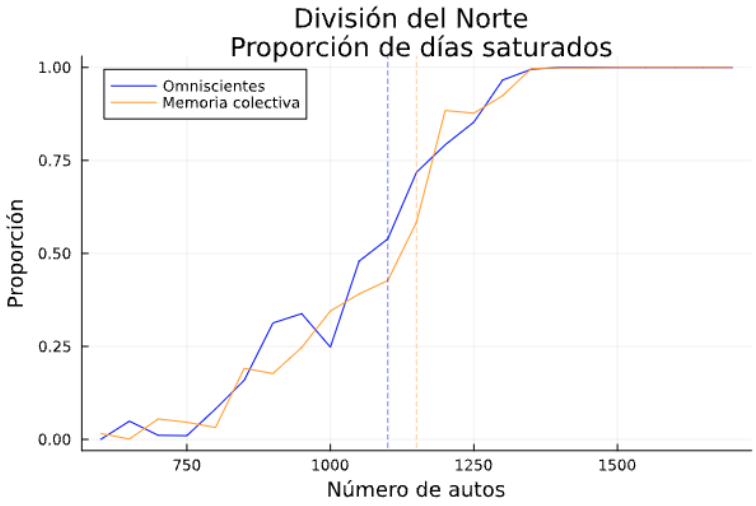

Figura 7.18: Saturación según el tipo de memoria; Red con diagonal tipo División del Norte.

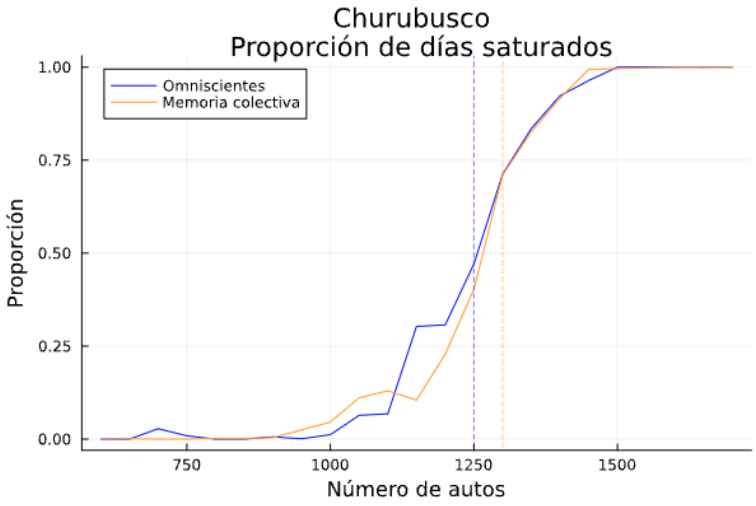

Figura 7.19: Saturación según el tipo de memoria; Red con diagonal tipo Churubusco.

La razón principal por la que se hicieron los dos tipos de memoria fue para analizar si era posible considerar el caso con conductores omniscientes como un estado más cercano al óptimo y utilizarlo como denominador en un cálculo del precio de la anarquía, sin embargo como la optimización sigue siendo individual y egoísta (cada conductor optimiza únicamente el costo de su propio camino), los resultados son similares (Imágenes: 7.17, 7.18 y 7.19), independientemente de que en un caso los conductores tienen más información según su propia experiencia y en el otro caso no.

La arquitectura con mayores diferencias es la de la red sin diagonal, posiblemente porque en dicha red no hay un camino hipotéticamente absolutamente mejor que los demás en costo.

7.4. Pruebas de integridad de las simulaciones

Se realizaron una serie de verificaciones para determinar que las simulaciones se mantuvieran dentro de parámetros esperados en una red de tráfico así como que no ocurrieran errores en los cálculos de parámetros.

Parte del análisis se realizó visualmente con animaciones de la dinámica de los automóviles y después se corroboró con los arreglos de velocidad, tiempos y posiciones de los automóviles, siguiendo este procedimiento se verificó que:

1. Los automóviles mantienen el orden a través de las aristas, es decir que no se pueden rebasar, esto tiene sentido en las redes que utilizamos, que son de un sólo carril.

2. Todos los automóviles tienen tiempos de llegada, distancias recorridas y velocidades promedio positivos, a excepción de los que no alcanzan su destino, en cuyo

caso las velocidades y tiempos tienen valores nulos.

3. Ningún auto puede viajar por fuera de las aristas de la digráfica y no pueden viajar en sentido contrario al de las calles. Esto incluye que los autos no pueden "pasarse de una esquinaz después volver.

4. Las simulaciones sólo terminan con una de dos condiciones: Todos los autos alcanzan su destino o no hay ningún movimiento posible por el estado de saturación de las aristas.

Otra comprobación importante es sobre las heurísticas utilizadas, a recordar: A* individual, A* con memoria colectiva, A* para conductores omniscientes. los costos son siempre admisibles (puesto que no es posible que los automovilistas recorran la red a través de las diagonales), pero no son en todos los casos consistentes (esencialmente esta condición depende de las velocidades recordadas en diferentes puntos de la red), A* admisible nos permite asegurar que A* termina y que obtiene caminos óptimos, aún si no es abriendo el menor número de nodos posible [13].

7.5. Resultados fallidos

En esta sección se discutirán algunos problemas que se encontraron durante el desarrollo de este trabajo y a partir de los cuales se debieron hacer correcciones a las simulaciones:

- Autos que se pasan de las esquinas y vuelven: Cuando se añadió la restricción de que los autos no podían salir o cambiar a una arista que tuviera excedida su capacidad, se observaba este fenómeno, porque los automovilistas seguían a su misma velocidad, por lo que cuando sucedía un paso discreto de tiempo podían exceder la longitud de la arista por la que viajaban. Esto se notó en animaciones realizadas y posteriormente se comprobó el problema al analizar arreglos de posiciones, el problema fue resuelto indicando un avance especial para este tipo de autos, indicando siempre que están "a punto"de alcanzar la siguiente arista de su camino.

- Conductores con mismo periodo de memoria individual: Cuando se comenzó a realizar simulaciones con autos, todos los agentes tenían el mismo número de días de memoria a guardar: 7, esto ocasionó que hubiese gráficas con comportamientos periódicos con periodo de 7, en tiempos, velocidades y autos cambiando, como se observa en la imagen 7.20. A partir de que notamos esto es que se decidió que cada auto tuviera un periodo diferente de memora, algo que de cualquier forma es más realista y que además decidiéramos incluir una componente colectiva de la memoria.

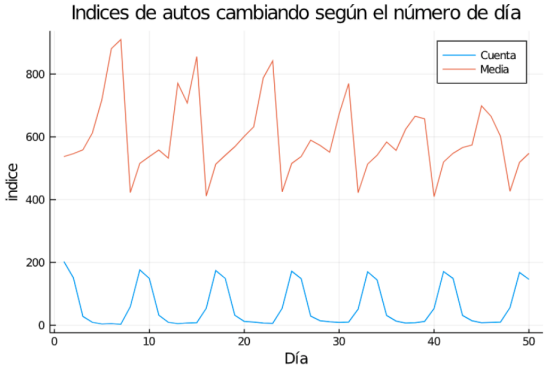

Figura 7.20: Ciclos en periodos de 7 días debido al periodo de memorias de los conductores.

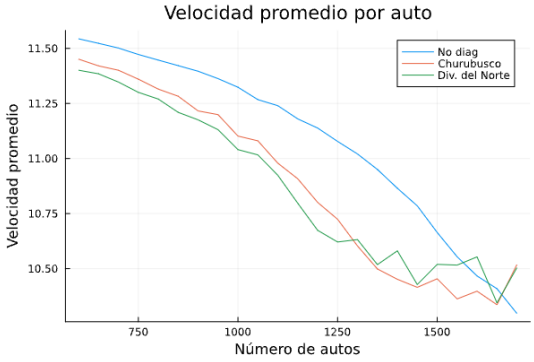

Figura 7.21: Velocidad promedio eliminando los valores nulos del cálculo de la velocidad; Memoria colectiva.

- Cálculo de tiempos y velocidades promedio omitiendo los valores nulos: Cuando se comenzaron a calcular los valores de simulaciones a través de varios días, se descubrió que los tiempos parecían subestimados y las velocidades sobreestimadas (imagen 7.21), debido a que sólo se consideraban los valores solamente de los autos que habían alcanzado su destino, esto quiere decir que podríamos calcular valores demasiado optimistas al no considerar el número de autos que nunca llegan a su destino. Para corregir esto decidimos asignar a los valores nulos el mínimo de los valores que sí están disponibles para las velocidades o el máximo de los valores disponibles para los tiempos.

Conclusiones

En este trabajo se hicieron simulaciones en redes viales cuadradas en tres diferentes escenarios: Una red cuadrada simple, una red cuadrada con una diagonal donde los automóviles deben detenerse cada vez que cruzan una calle y otra parecida a un distribuidor vial, donde los automóviles deben esperar para ingresar pero una vez sobre ella no deben detenerse.

Notamos que en ambos casos, adicionar una diagonal resulta benéfico cuando la densidad de autos es baja, pero inesperadamente, ambas alteraciones resultan contraproducentes cuando la densidad de autos es medianamente alta. Esto, aunque similar a la paradoja de Braess (*"A partir de un sistema donde la toma de decisiones de rutas es egoísta, una mejora en la red puede degradar el desempeño de la red."*), resulta más bien una extensión de la misma, para un caso donde los automovilistas no salen de un único punto para llegar a un único destino, sino que sus orígenes y destinos están homogéneamente distribuidos en la red. Además de que se consideran salidas de los automóviles en diferentes puntos del tiempo.

En las simulaciones medimos varias cantidades relacionadas con el estado del tráfico en la red: Específicamente velocidad promedio en el recorrido, tiempo promedio en el recorrido, distancia recorrida del origen al destino, número de automóviles atascados y posiciones de las aristas saturadas. Todas las cantidades relacionadas a la eficiencia del tráfico, parecen empeorar abruptamente al rededor de un valor crítico de la densidad de autos (lo que significa una reducción en la velocidad y un aumento en los tiempos promedio), que es menor que la capacidad de la red.

El valor crítico de la saturación de la red relacionado al número de automóviles liberados en cada simulación (densidad vehicular) se reduce cuando se agregan las diagonales. Atribuimos este fenómeno al menos a un par de causas:

- La ruta utilizando la diagonal es más atractiva en costo que las alternativas para muchos de los conductores, es decir que cualquier conductor que pueda beneficiarse de ella, lo hará. A su vez, el incremento en el flujo de la diagonal provocará cuellos de botella sobre la misma, en sus entradas y en sus salidas. Lo que impacta incluso a conductores que deciden no utilizarla.
- La diagonal incrementa el número de intersecciones y por ello, se reduce la capacidad de carga de algunas aristas, recuérdese que es una regla que los automóviles

reduzcan su velocidad antes de cada intersección.

Para poner a prueba la primera hipótesis sería deseable poder medir el precio de la anarquía de las redes con diversas arquitecturas; sin embargo, el modelo que propusimos, por su complejidad, limita la posibilidad de realizar mediciones analíticas de flujo en un caso ideal. Sería deseable entonces encontrar alguna forma de al menos aproximar el flujo ideal con una modificación a las reglas de cada uno de los agentes de la simulación. Para esto, intentamos utilizar conductores omniscientes, pero el resultado no fue fructífero ya que, a pesar de que los conductores tienen una memoria compartida del estado de la red, las decisiones que toman, incluso por diseño, buscan optimizar únicamente el costo de su propia ruta. Lo que provoca que incluso conforme la diagonal se vuelve costosa, deciden tomarla una y otra vez, con el egoísmo de buscar sólo el beneficio propio y quizás con el optimismo de que esta vez no estará tan llena.

Hay varios temas que proponemos como posibles extensiones de este trabajo, que resumimos a continuación:

- Modificaciones con distribuciones del parámetro k para el perfil de riesgo de los conductores
- Distribuciones diferentes de velocidad
- Diferentes tamaños de red
- Diferentes topologías de red
- Pruebas con diversos tipos de combinaciones de memorias individuales y colectivas
- Pruebas con distribuciones específicas de orígenes y destinos

Finalmente cabe mencionar el valor de las densidades críticas que encontramos en este trabajo, es posible que estas densidades existan en diferentes tamaños y arquitecturas de red. Y como se vio, es la saturación lo que causa cambios importantes en los parámetros de eficiencia de la red.

Para sistemas reales, sería muy valioso analizar qué tan cercanos de esas densidades críticas nos encontramos para poder proponer soluciones al problema de tráfico que tanto demerita la calidad de vida de los habitantes. Para el caso particular de la Ciudad de México es muy posible que eliminar diagonales, pueda, paradójicamente, aliviar una parte del problema.

Cabe mencionar como último comentario, que creemos fervientemente que la migración hacia otros sistemas de transporte, menos contaminantes y colectivos, y la consideración al prójimo (en el cálculo de costos de rutas, pero en muchísimos más ámbitos) podría ser la solución fundamental.

Referencias

[1] Abadi, A., Rajabioun, T., and Ioannou, P. A. (2014). Traffic flow prediction for road transportation networks with limited traffic data. *IEEE transactions on intelligent transportation systems*, 16(2):653–662. 29

[2] Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. (1993). *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall. 8, 43, 44

[3] Besançon, M. (2018). Lightgraphsflows package. `https://github.com/JuliaGraphs/LightGraphsFlows.jl`. [Visitado el 08/Junio/2023]. 69

[4] Bondy, J. A. and Murty, U. S. R. (2008). *Graph Theory*. Springer. 4, 7

[5] Braekers, K., Ramaekers, K., and Van Nieuwenhuyse, I. (2016). The vehicle routing problem: State of the art classification and review. *Computers & industrial engineering*, 99:300–313. 29

[6] Brass, P. (2008). *Advanced Data Structures*. Cambridge University Press. 30, 31

[7] Brassard, G. and Bratley, P. (1996). *Fundamentals of Algorithmics*. Prentice-Hall, Inc., USA. 7, 25, 26, 28, 29, 30

[8] Chartrand, G. and Zhang, P. (2009). *Chromatic Graph Theory*. Taylor and Francis Group, LLC, 2nd edition. 4

[9] Cominetti, R., Facchinei, F., Lasserre, J., Daniilidis, A., and Martínez-Legaz, J. (2012). *Modern Optimization Modelling Techniques*. Advanced Courses in Mathematics - CRM Barcelona. Springer Basel. 14

[10] Cormen, T. H. e. a. (2009). *Introduction to algorithms*. MIT Press, Cambridge, MA. 19, 24, 25, 26, 28, 29

[11] Dasgupta, S., Papadimitriou, C. H., and Vazirani, U. V. (2016). *Algorithms*. McGraw-Hill Education (India). 23, 25

[12] Diao, Z., Zubairy, M. S., and Chen, G. (01 Aug. 2002). A quantum circuit design for grover's algorithm. *Zeitschrift für Naturforschung A*, 57(8):701 – 708. 19

[13] Edelkamp, S. and Schroedl, S. (2011). *Heuristic search: theory and applications*. Elsevier. 7, 8, 26, 27, 33, 86

[14] Estrada, E. (2015). *A First Course in Network Theory*. Oxford University Press. 3

[15] Fairbanks, J., Besançon, M., Simon, S., Hoffiman, J., Eubank, N., and Karpinski, S. (2021). Juliagraphs/graphs.jl: an optimized graphs package for the julia programming language. [Visitado el 05/Junio/2023]. 48

[16] INEGI (2020). Censo de población y vivienda. https://www.inegi.org.mx/programas/ccpv/2020/. [Visitado el 29/Abril/2022]. 1

[17] INEGI (2021). Parque vehicular. https://www.inegi.org.mx/temas/vehiculos/. [Visitado el 29/Abril/2022]. 1

[18] John Forrest, David de la Nuez, R. L.-H. (2004). Clp package. https://github.com/coin-or/Clp. [Visitado el 08/Junio/2023]. 69

[19] Lárraga, M. E. and Alvarez-Icaza, L. (2010). Cellular automaton model for traffic flow based on safe driving policies and human reactions. *Physica A: Statistical Mechanics and its Applications*, 389(23):5425–5438. 14

[20] Lenstra, J. K. and Kan, A. R. (1981). Complexity of vehicle routing and scheduling problems. *Networks*, 11(2):221–227. 29

[21] Newell, G. F. (1961). Nonlinear effects in the dynamics of car following. *Operations research*, 9(2):209–229. 14

[22] Newman, M. E. J. (2010). *Networks: An Introduction*. Oxford University Press. 4, 7

[23] Roughgarden, T. (2005). *Selfish Routing and the Price of Anarchy*. MIT Press. 15, 16, 17

[24] Roughgarden, T. (2017a). *Algorithms Illuminated (Part 1): The Basics*. Algorithms Illuminated. Soundlikeyourself Publishing, LLC. 21

[25] Roughgarden, T. (2017b). *Algorithms Illuminated (Part 2): Graph Algorithms and Data Structures*. Algorithms Illuminated. Soundlikeyourself Publishing, LLC. 30

[26] Roughgarden, T. and Tardos, É. (2002). How bad is selfish routing? *Journal of the ACM (JACM)*, 49(2):236–259. 17

[27] Russell, S. and Norvig, P. (2010). *Artificial intelligence: a modern approach*. Pearson education, 3rd edition. 33

[28] Sayama, H. (2015). *Introduction to the Modeling and Analysis of Complex Systems*. Open SUNY Textbooks. 13

[29] Seth Bromberger, J. F. and other contributors (2017). Juliagraphs/lightgraphs.jl: an optimized graphs package for the julia programming language. [Visitado el 13/Mayo/2023]. 57

[30] Simon Kornblith, T. H. (2022). Jld package. https://github.com/JuliaIO/JLD.jl. [Visitado el 01/Junio/2023]. 64

[31] Singh, R. and Dowling, R. (2002). Improved speed-flow relationships: application to transportation planning models. In *Seventh TRB Conference on the Application of Transportation Planning MethodsTransportation Research Board; Commonwealth of Massachusetts, Executive Office of Transportation and Construction; and Boston Metropolitan Planning Organization*. 18

[32] Susskind, L. (2016). Addendum to computational complexity and black hole horizons. *Fortschritte der Physik*, 64(1):44–48. 23

[33] TomTom, T. (2019). Traffic congestion ranking: Tom tom traffic index. [Visitado el 29/Abril/2022]. 1

[34] Treiber, M., Hennecke, A., and Helbing, D. (2000). Congested traffic states in empirical observations and microscopic simulations. *Physical review E*, 62(2):1805. 14

[35] Vajda, S. (2008). *Fibonacci and Lucas numbers, and the golden section: theory and applications*. Courier Corporation. 22

[36] Van Wageningen-Kessels, F., Hoogendorn, S., Vuik, K., and van Lint, H. (2015). Traffic and transportation simulation: Looking back and looking ahead: Celebrating 50 years of traffic flow theory, a workshop. In *Transportation Research Circular EC195*. Publisher. 14

[37] Wang, X., Xiao, N., Xie, L., Frazzoli, E., and Rus, D. (2015). Analysis of price of anarchy in traffic networks with heterogeneous price-sensitivity populations. *IEEE Transactions on Control Systems Technology*, 23(6):2227–2237. 17

[38] Youn, H., Gastner, M. T., Jeong, H., and Park, H. (2008). Price of anarchy in transportation networks: Efficiency and optimality control. *Physical Review Letters*, 101(12):128701. 16

[39] Zhang, J., Pourazarm, S., Cassandras, C. G., and Paschalidis, I. C. (2018). The price of anarchy in transportation networks: Data-driven evaluation and reduction strategies. *Proceedings of the IEEE*, 106(4):538–553. 17