MEMORIA - PRÁCTICA 3
Práctica mensajería

Fecha de entrega:
22 - Diciembre - 2023

Autores y NIP:
Óscar Brizuela García (820773)
Pilar Fierro Casanova (819658)

Profesores:
Elvira Mayordomo
Ubaldo Ramón

Grado:
Cuarto curso de Ingeniería Informática

Asignatura:
Algoritmia para problemas difíciles

Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Universidad
Zaragoza

RESUMEN

Este resumen recoge el trabajo realizado para llevar a cabo la tercera práctica de la asignatura de Algoritmia para Problemas Difíciles, simulando el servicio de mensajería entre dos empresas. Para ello se calcula el tiempo de entrega de un paquete mediante un algoritmo probabilista que considera la localización de los almacenes de dos empresas, la localización de la casa del comprador, las intersecciones para llegar al destino y las carreteras que unen dichas intersecciones. El código fuente entregado consta de dos ficheros: *graph_generator.py,* con el código en *Python* para generar los datos de entrada, y *main.cc,* con la implementación del programa principal.

En este documento se incluye un apartado de implementación donde se cuenta más en detalle las funciones para la generación de grafos y la gestión de las probabilidades, así como para el programa principal que se encarga de simular la entrega de paquetes generado.

Por otro lado se cuenta con una sección de pruebas donde se recogen las distintas pruebas realizadas habiendo una clara distinción entre grafos conexos y no conexos, para los distintos casos con un posterior análisis y conclusión de éstos.

ÍNDICE

1. INTRODUCCIÓN

En esta práctica se ha realizado una simulación donde dos empresas ofrecen un servicio de mensajería. Se escogerá aquella cuyo tiempo estimado de entrega del paquete sea menor, en función de la localización de la casa de la persona que realiza el pedido y de los almacenes de ambas empresas, cuyas localizaciones son distintas entre sí.

Para la entrega de un paquete se emplea en ambos casos la misma estrategia:

- ➢ Un conductor sale de un almacén que está directamente conectado a una intersección.
- ➢ Cuando llega a la intersección se elige una carretera aleatoriamente (dada una distribución de probabilidades conocida de las carreteras que salen de la intersección).
- ➢ Si el conductor llega a la intersección donde vive el comprador, entrega el paquete o lo deja en la puerta en caso de que no esté.
- ➢ Se deben tener en cuenta los siguientes aspectos:
    - Cada intersección está conectada a una carretera que, a su vez, está conectada a una intersección distinta.
    - Recorrer una carretera no lleva un tiempo fijo: el tiempo, descrito en minutos, es propio de la carretera.

Para poder llevar a cabo esta simulación cumpliendo con la estrategia descrita, se ha implementado un programa repartido en dos ficheros y empleando dos lenguajes de programación distintos que se detallan a continuación a lo largo del documento. Se trata de simular que "atraviesa" el mapa desde cada almacén hasta la casa donde se ha de entregar el paquete, calculando el tiempo de entrega. Este será la suma de los tiempos de todas las carreteras recorridas.

2. CÓDIGO FUENTE

2.1. Ficheros entregados

El código fuente entregado para resolver la práctica se ha distribuido en dos ficheros: *graph_generator.py* y *main.cc*.

En el primero de estos se encuentra el código que genera un fichero *.txt,* llamado *generated_graph.txt*, con los datos de entrada cada vez que se ejecuta. Se ha de tener en cuenta que cada escenario (el propio grafo generado en el fichero *.txt*) debe cumplir con las restricciones descritas en el enunciado de la práctica.

Por otro lado, el fichero *main.cc* contiene las funciones del programa principal que se encargan de generar la simulación de las entregas de los paquetes con los datos que se obtienen el fichero de entrada *generated_graph.txt.*

2.2. Comandos para ejecutar

El orden de los comandos para la correcta ejecución de los ficheros fuente entregados es el siguiente:

- En primer lugar, se ejecuta `python graph_generator.py`, para generar el fichero *.txt* con los datos de entrada.
- En segundo lugar, una vez se tiene el fichero *.txt* con los datos de entrada correspondientes a una simulación, se compila el programa principal mediante `g++ main.cc -o main,` y posteriormente se ejecuta mediante `./main`.

3. IMPLEMENTACIÓN

A continuación se detalla más en profundidad la implementación del código fuente entregado.

3.1. Generación de datos de entrada

El fichero *graph_generator.py*, implementado en *Python*, genera un grafo con una configuración determinada y obteniendo como resultado de ejecución un fichero de salida *generated_graph.txt*. En él se encuentran los datos pertenecientes al grafo de un posible escenario que representa un mapa, con sus respectivas intersecciones y carreteras.

En primer lugar se encuentran unas funciones correspondientes a la generación y obtención de información de los grafos. Cabe destacar que los grafos son representados como listas de tuplas de dos elementos. De esta forma, cada tupla *(a, b)* representa que existe una arista (carretera) en el grafo que va desde el vértice (intersección) *A* hasta el vértice *B*.

El grafo de la siguiente figura, por ejemplo, está descrito con la siguiente lista de tuplas: E={{v1, v3}, {v1, v4}, {v2, v3}, {v2, v4}, {v3, v5}, {v4, v5}}. En este caso se habla de un grafo conexo, puesto que no existen subgrafos ni vértices aislados, pero no completo, ya que hay ciertos vértices que no están conectados entre sí (como *v1* y *v2*).

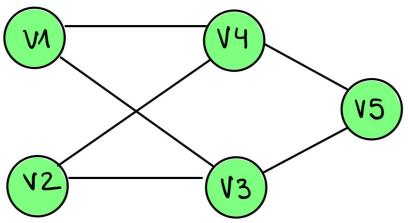

Figura 1: grafo conexo

Las funciones necesarias para crear estos grafos (listas de tuplas) son las siguientes:

- *random_connected_graph(n, m)*: devuelve una lista de aristas de un grafo conexo generado aleatoriamente conectado *n* vértices mediante *m* aristas. El grafo generado deberá tener mínimo n - 1 aristas, ya que así se garantiza que no existe ningún vértice que no esté conectado con el grafo, o la generación de subgrafos desconexos entre sí.

- *random_non_connected_graph(n, m)*: devuelve una lista de aristas de un grafo generado aleatoriamente, que no tiene por qué ser conexo, de *n* vértices y *m* aristas. En este caso, por tanto, es posible que existan vértices aislados o subgrafos que no estén conectados entre sí.

- *cycle_graph_edges(n)*: devuelve las aristas de un grafo cíclico generado de *n* vértices. Así, este grafo representa una forma de polígono de *n* vértices. Cada vértice, por tanto, solo cuenta con dos aristas.

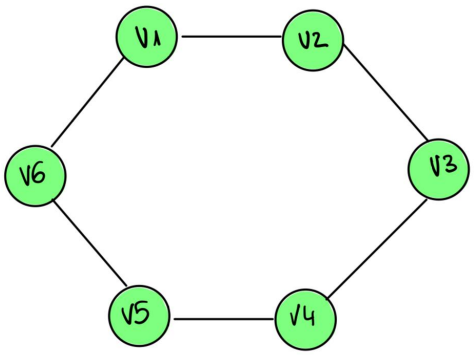

Figura 2: grafo cíclico de 6 vértices

- *star_graph_edges(n)*: devuelve las artistas de un grafo en estrella generado de *n* vértices. Por tanto, en este grafo existirá un vértice central, conectado a todos los demás. Sin embargo, aquellos vértices que no sean el central (n - 1) solo estarán conectados al vértice central, creando una estructura de estrella.

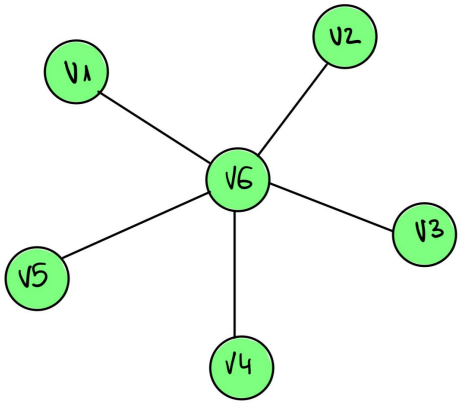

Figura 3: grafo en estrella de 6 vértices

- ➢ *remove_duplicate_edges(edges)*: devuelve una lista de aristas habiendo sido eliminadas aquellas que se encontraban duplicadas.
- ➢ *sort_tuples(lst)*: dada una lista de aristas, devuelve dicha lista ordenada en función de los elementos de las tuplas.
- ➢ *get_unique_vertices(graph)*: obtiene la lista de vértices únicos en el grafo proporcionado por parámetro.
- ➢ *count_vertex_occurrences(graph)*: cuenta el número de ocurrencias de cada vértice en un grafo proporcionado por parámetro, devolviendo un diccionario donde las claves son los vértices y los valores son las ocurrencias correspondientes.

Por otro lado, se encuentran las funciones de gestión de probabilidades. Estas tienen como principal objetivo asegurar que la suma de las probabilidades de las carreteras que salen de una intersección (aristas que salen de un vértice) sumen 1, puesto que el repartidor puede decidir por una u otra de estas carreteras.

- ➢ *calculate_vertex_probabilities(vertices, occurrences, graph)*: devuelve un diccionario con las probabilidades de cada arista que parte de cada vértice de un grafo en función de sus ocurrencias.
- ➢ *extract_edge_probabilities(input_dict)*: dado un diccionario con las probabilidades de cada arista de cada vértice según las ocurrencias, devuelve otro diccionario en el que el formato de las aristas con las probabilidades ya es el adecuado. Las claves del diccionario son el par de vértices *(a, b)*, y los valores del diccionario son las probabilidades correspondientes de ir de un vértice a otro (de *a* a *b*).

- ➢ *get_probabilities_for_vertex(graph, vertex)*: obtiene las probabilidades asociadas a un vértice en un grafo.

Seguidamente, ha sido necesaria la implementación de una función que, dado un grafo y dos vértices de dicho grafo, devuelve un *booleano* que indica si existe un camino entre dichos vértices en ese grafo. Esta función, llamada *has_path(graph, vertice1, vertice2)*, ha sido necesaria para asegurar que en los grafos no conexos, donde pueden existir distintos subgrafos desconectados entre sí pero pertenecientes al mismo grafo, los vértices correspondientes a los almacenes de las empresas *A* y *B* y el correspondiente a la casa del comprador (C) siempre se encuentren en el mismo subgrafo. De esta manera, se garantiza que siempre exista un camino desde cualquiera de los almacenes hasta la casa, puesto que no se encuentran en subgrafos desconectados.

Por último, las funciones siguientes se encargan de describir correctamente los datos de entrada que representan el grafo en el formato especificado por el guión de la práctica (cada arista conformará una línea del fichero, con las probabilidades correspondientes y el tiempo en minutos para recorrer dicha arista):

- ➢ *insert_random_number(lst)*: inserta números aleatorios en cada arista de la lista (minutos de la carretera).
- ➢ *generate_line_connected_or_not(output_filename)*: genera una línea con valores aleatorios para crear un grafo que está conectado o no.
- ➢ *write_to_file(output_filename, lst)*: escribe las aristas en un archivo.

También se ha tenido en cuenta que a la hora de generar un fichero *.txt* el almacén, es decir, donde se comienza, no se encuentre en la misma intersección donde se ha de entregar el paquete. Los dos almacenes tampoco pueden estar en la misma intersección, por lo que el vértice correspondiente al almacén de la empresa *A* no puede ser el mismo que el correspondiente a la empresa *B*.

3.2. Algoritmo probabilista

Este algoritmo, implementado en el fichero *main.cc*, devuelve el tiempo en minutos que tarda un repartidor que sale desde un almacén hasta que llega a la casa del comprador, dada una matriz de datos de tipo *double* que contiene las probabilidades de ir de un vértice cualquiera a otro del grafo y otra matriz que indica el tiempo que se tarda en recorrer la carretera que conecta dos intersecciones cualesquiera del grafo (vértices).

Esta función toma como datos de entrada, por tanto, las matrices *mapa_probabilidades* y *mapa_tiempos*, el número de vértices *n_vertices*, el destino *casa* y el origen *almacen_empresa*.

Debido a que los grafos generados con el programa *graph_generator.py* aseguran que el almacén de una empresa no puede estar en la misma intersección que la casa del comprador (ambos vértices han de ser distintos), el repartidor deberá recorrer como mínimo una carretera (en caso de que el almacén y la casa estén conectados directamente, y el repartidor escoja dicha carretera al comenzar).

El camino que va recorriendo el repartidor desde el almacén correspondiente a la casa del comprador se va almacenando en la variable *camino*, que consta de un vector en memoria dinámica de datos de tipo *int* para guardar aquellas intersecciones (vértices) visitadas.

El repartidor va recorriendo intersecciones hasta que, evidentemente, se encuentre en aquella correspondiente a la casa del comprador, y pueda entregar el paquete. Por tanto, una vez el repartidor se encuentra en una intersección concreta, el repartidor crea una lista con todas las intersecciones a las que puede llegar desde dicha intersección (todos los vértices que están directamente conectados al vértice donde se encuentra en ese momento). Además, en esta lista "apunta" también las probabilidades de ir por cada una de las intersecciones, puesto que es más probable ir a unas intersecciones determinadas que a otras. Estas probabilidades se extraen del parámetro *mapa_probabilidades* que se le ha pasado al algoritmo probabilista.

Una vez el repartidor cuenta con la lista de las intersecciones a las que puede llegar desde el punto donde se encuentra en ese momento y las probabilidades de llegar a cada una de estas intersecciones (cada una de las cuales será siempre mayor que 0), se hace uso de la función *getRandomNonZeroIndex*(). Esta función acepta como parámetro un vector de enteros, donde los índices son las intersecciones (vértices) a las que el repartidor puede llegar desde donde está y los valores de dichos índices son las probabilidades de que estos índices sean devueltos. Así, aquellos índices del vector con mayor valor (debe estar en el intervalo (0, 1], pues es una probabilidad) tendrán mayor probabilidad de ser elegidos por el repartidor. En otras palabras, el repartidor llegará a las intersecciones en función de la probabilidad que existe entre el la intersección donde se encuentra y las posibles intersecciones destino.

Esta funcionalidad se implementa mediante una técnica matemática llamada probabilidad acumulada que, a pesar de que no va a ser explicada en este documento, asegura que los índices del vector con un valor de probabilidad más alto sean devueltos más frecuentemente que aquellos índices con un valor de probabilidad más bajo.

Cuando se llega a la nueva intersección, se comprueba si esta corresponde a la casa del comprador. Si es así, el repartidor entrega el paquete, por lo que el flujo del programa sale del bucle. Si no, repite el proceso comentado previamente desde la nueva intersección en la que se encuentra.

Una vez el paquete ha sido entregado, se hace uso finalmente del parámetro *mapa_tiempos* que se le ha pasado inicialmente al algoritmo probabilista. La variable *tiempo_camino* se actualiza con los tiempos de recorrer aquellas carreteras (aristas entre dos vértices del grafo) que se han ido recorriendo, pues dichas carreteras se han ido guardando en el vector de enteros camino. Por tanto, se visitan aquellos elementos de la matriz *mapa_tiempos* en función del orden de las intersecciones recorridas guardadas en camino. El algoritmo devuelve, finalmente, la variable *tiempo_camino*, que tiene el valor de los tiempos, acumulados, correspondientes a las carreteras recorridas durante el reparto.

3.3. Bootstraps

Para la generación de los 500 bootstraps para cada empresa se han seguido los siguientes pasos:

- Generación de las simulaciones: el fichero *main.cc* contiene el siguiente bucle, donde se generan *n* simulaciones para cada una de las empresas con la ayuda del algoritmo *probabilistic_algorithm()* descrito en el apartado anterior. Se guardan los tiempos correspondientes a cada una de las entregas de ambas empresas.

```
for(int i = 0; i < n; i++) {
    simulaciones_A[i] = probabilistic_algorithm(mapa_probabilidades, mapa_tiempos, linea.N, linea.C, linea.A);
    simulaciones_B[i] = probabilistic_algorithm(mapa_probabilidades, mapa_tiempos, linea.N, linea.C, linea.B);
}
```

Figura 4: fragmento de código que genera las n simulaciones para las empresas A y B

- Obtención de las medias de las simulaciones: se obtiene también una media de los tiempos correspondientes a las simulaciones de cada una de las empresas. La obtención de estas medias se encuentra dividida en dos bloques de código similares, cada uno para una empresa de mensajería:

```cpp
// Calcular media simulaciones almacen A
int suma_simulaciones_A = 0;
double media_simulaciones_A = 0.0;
for (int i = 0; i < n; i++) {
    suma_simulaciones_A += simulaciones_A[i];
}
media_simulaciones_A = static_cast<double>(suma_simulaciones_A) / n;

// Calcular media simulaciones almacen B
int suma_simulaciones_B = 0;
double media_simulaciones_B = 0.0;
for (int i = 0; i < n; i++) {
    suma_simulaciones_B += simulaciones_B[i];
}
media_simulaciones_B = static_cast<double>(suma_simulaciones_B) / n;
```

Figura 5: fragmento de código de la función main que realiza la media del coste de las simulaciones

- Generación de bootstraps y obtención de sus medias: para realizar la parte correspondiente a los bootstraps se realizan dos bucles de 500 iteraciones, uno para cada empresa, donde se seleccionan aleatoriamente *n* valores con reemplazo de las simulaciones originales. Se obtienen también las medias correspondientes a los bootstraps.

```cpp
// Calcular bootstraps almacen A
for (int i = 0; i < 500; i++) {
    int suma_simulaciones_A_bootstrap = 0;
    double media_simulaciones_A_bootstrap = 0.0;

    // Elegir aleatoriamente n valores con reemplazo
    for (int j = 0; j < n; j++) {
        int randomIndex = rand() % n;
        simulaciones_A_bootstrap[j] = simulaciones_A[randomIndex];
    }

    // Calcular media de esos valores
    for (int i = 0; i < n; i++) {
        suma_simulaciones_A_bootstrap += simulaciones_A_bootstrap[i];
    }
    media_simulaciones_A_bootstrap = static_cast<double>(suma_simulaciones_A_bootstrap) / n;

    // Cada bootstrap guardará una media
    bootstraps_A[i] = media_simulaciones_A_bootstrap;
}
```

Figura 6: fragmento de código con generación de bootstraps para la empresa A ubicado en el main principal del programa

- Ordenación de las medias y obtención de los intervalos de confianza: estas medias se ordenan en orden creciente, y se atiende a las fórmulas descritas en el enunciado para

adquirir los valores que delimitan los intervalos de confianza al 90% para cada una de las empresas.

```
// Ordenar las medias de cada bootstrap
sort(bootstraps_A, bootstraps_A + 500);
sort(bootstraps_B, bootstraps_B + 500);


// Intervalos [L, R]
double R_A = 2 * media_simulaciones_A - ((bootstraps_A[24] + bootstraps_A[25]) / 2);
double L_A = 2 * media_simulaciones_A - ((bootstraps_A[474] + bootstraps_A[475]) / 2);

double R_B = 2 * media_simulaciones_B - ((bootstraps_B[24] + bootstraps_B[25]) / 2);
double L_B = 2 * media_simulaciones_B - ((bootstraps_B[474] + bootstraps_B[475]) / 2);
```

Figura 7: fragmento de código con ordenación de las medias de los bootstraps y obtención de los intervalos de confianza para ambas empresas

4. PRUEBAS

Como se ha comentado anteriormente, la realización de las pruebas se ha llevado a cabo mediante la terminal de Visual Studio Code ejecutando los siguientes comandos: `python graph_generator.py`, para generar el fichero con el grafo que contiene los datos de entrada y posteriormente: `g++ main.cc -o main`, `./main` para ejecutar el main principal del programa que lee dicho fichero y lo analiza simulando las correspondientes entregas de un paquete.

Las pruebas se encuentran divididas en este apartado según el tipo de grafo utilizado. Cada una de ellas tiene una serie de valores N (número de vértices), para cada uno de los cuales existe un valor o conjunto de valores M (aristas), dependiendo del tipo de grafo.

Cada una de las tablas guarda información sobre los valores de los intervalos de confianza obtenidos y los tiempos de ejecución. Como criterio para decidir qué empresa se debe escoger en cada caso, actuando como si fuéramos el comprador, se ha decidido considerar aquella empresa cuyo intervalo completo sea menor en la recta de los reales. Esto implica que los dos valores que definición su intervalo, tanto el valor de L como el de R, han de ser menores que los de su empresa competidora. Así, si existe una intersección entre ambos intervalos (el valor de L de la empresa A es menor que el de L de la empresa B, pero el de R de la empresa B es menor que el de R de la empresa A, por ejemplo), la empresa que debería escogerse en ese caso queda indefinida (“No se sabe” en la columna de Mejor empresa” de las tablas).

Las tablas también permiten visualizar, en microsegundos, los tiempos de ejecución correspondientes a los 500 *boostraps*.

4.1. Grafos conexos

- Valores de N: 10, 150, 400 y 600.
- Valores de M (en función de N): $\frac{(N*(N-1))/2 + N/2}{2}$ y $\frac{N*(N-1)}{2}$

El primero de los valores de M (en función de N) de los grafos conexos utiliza el número "mediano" de aristas para un N dado, según las condiciones del enunciado. Así, se encuentra el punto medio entre el mínimo número de aristas para un grafo de N vértices según el enunciado (N/2, aunque correspondería a grafos no conexos) y el máximo número de aristas para un grafo de N vértices (((N * (N - 1))/2),

El segundo de los valores de M (en función de N) para un grafo conexo es el que conforma un grafo denominado "completo", puesto que todos sus vértices están unidos entre sí. Es imposible que un grafo de N vértices tenga más aristas que el indicado por este valor (suponiendo que, entre dos intersecciones, solo existe una única carretera directa que los une, es decir, dos vértices pueden estar unidos por una sola arista).

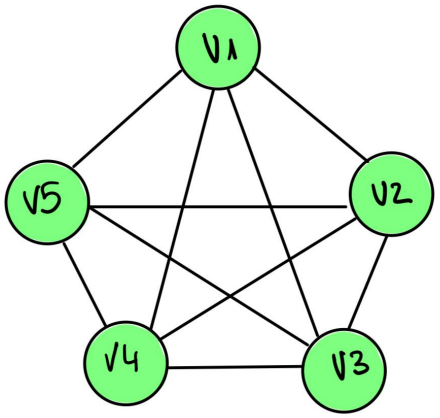

Figura 8: grafo completo de 5 vértices

N = 10								
	L_A	**R_A**	**Valor intervalo A**	**L_B**	**R_B**	**Valor intervalo B**	**Mejor opción**	**Tiempo ejecución** (en µs)
M = 25 Casa: 7 A: 6 B: 8	**94,985**	**136,81**	41,825	**128,07**	**169,245**	41,175	**Empresa A**	22762
M = 45 Casa: 2 A: 7	**84,49**	**126,11**	41,62	**95,065**	**127,985**	32,92	**Empresa A**	7997

B: 5								

Tabla 1: pruebas para grafo conexo con distintos valores de M para N = 10

N = 150								
	L_A	**R_A**	**Valor intervalo A**	**L_B**	**R_B**	**Valor intervalo B**	**Mejor opción**	**Tiempo ejecución** (en µs)
M = 5625 Casa: 72 A: 24 B: 137	**1129,27**	**1535,01**	405,74	**1649,44**	**2122,73**	473,295	**Empresa A**	337860
M = 11175 Casa: 97 A: 31 B: 82	**1421,82**	**1886,83**	465,01	**1392,02**	**2016,45**	624,435	**No se sabe**	370693

Tabla 2: pruebas para grafo conexo con distintos valores de M para N = 150

N = 400								
	L_A	**R_A**	**Valor intervalo A**	**L_B**	**R_B**	**Valor intervalo B**	**Mejor opción**	**Tiempo ejecución** (en μs)
M = 40000 Casa: 56 A: 255 B: 114	**3296,97**	**4612,85**	1315,88	**3265,51**	**4595,05**	1329,55	**Empresa B**	1880921
M = 79800 Casa: 218 A: 260 B: 233	**3198,07**	**4373,22**	1175,15	**3590,1**	**4869,45**	1279,35	**Empresa A**	1971264

Tabla 3: pruebas para grafo conexo con distintos valores de M para N = 400

N = 600								
s	**L_A**	**R_A**	**Valor intervalo A**	**L_B**	**R_B**	**Valor intervalo B**	**Mejor opción**	**Tiempo ejecución** (en μs)
M = 90000 Casa: 88 A: 398 B: 590								
M = 179700 Casa: 42 A: 110 B: 361								

Tabla 4: pruebas para grafo conexo con distintos valores de M para N = 600. Debido al alto número de aristas, nuestras máquinas no son capaces de resolverlo en un tiempo adecuado

4.2. Grafos no conexos

Para que un grafo sea conexo, el número de aristas m tiene que ser mínimo n - 1, siendo n el número de vértices. De esta forma, la función que asegura que todos los vértices estén conectados imposibilita que exista en el grafo creado algún vértice que quede aislado sin estar conectado con el resto del grafo.

Sin embargo, debido a que el enunciado permite que el número de aristas sea N/2 y se ha considerado este caso como relevante, se ha decidido implementar una función que pueda crear un grafo no conexo, con un número de aristas menor que n - 1. Así, el grafo creado sería no conexo, por lo que pueden existir varios subgrafos desconectados entre sí y/o vértices aislados.

Por otro lado, como ya se ha comentado en el punto X, se ha implementado una función que determina si existe un camino entre dos vértices del grafo. El objetivo de esta función es ser utilizada con los vértices correspondientes a las ubicaciones tanto de los almacenes de ambas empresas junto con el vértice correspondiente a la ubicación de la casa.

De esta forma, podemos crear un grafo que, a pesar de no ser conexo, contiene un subgrafo con los 3 vértices considerados (almacén de la empresa A, almacén de la empresa B y casa del comprador), de manera que exista un camino entre ellos.

La generación de este grafo se realiza mediante un bucle while: se genera un grafo no conexo y, si no existe un camino tanto desde el almacén de la empresa A hasta la casa del comprador como desde el almacén de la empresa B hasta la casa del comprador, se vuelve a generar otro grafo no conexo hasta que ambas condiciones se cumplan.

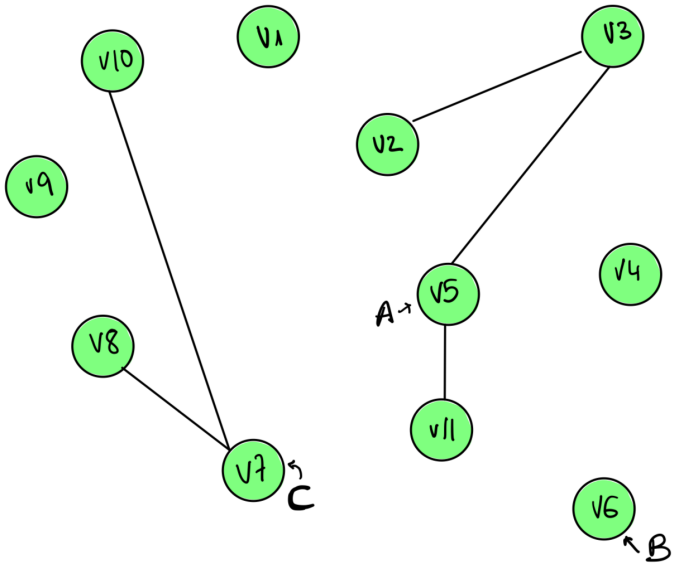

Figura 9: grafo no conexo donde los almacenes (A y B) están en subgrafos diferentes al que se encuentra la casa (C)

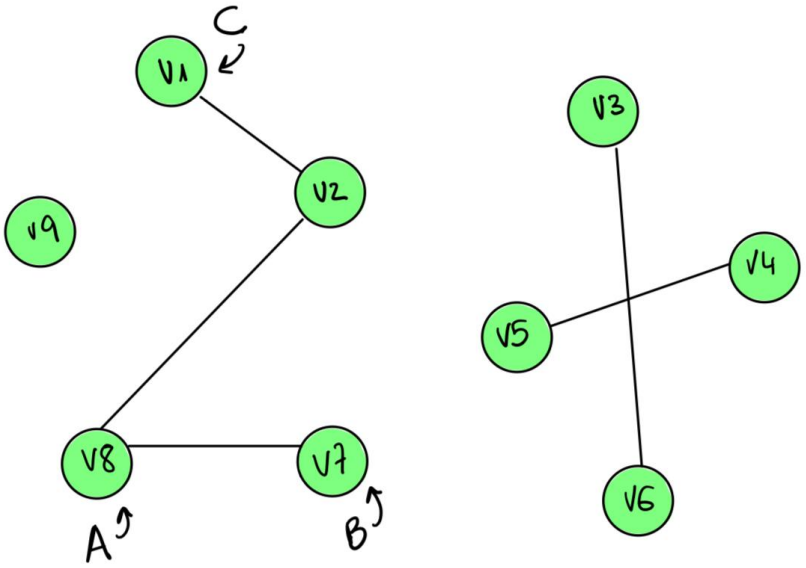

Figura 10: grafo no conexo donde ambos almacenes (A y B) se encuentran en el mismo subgrafo en el que está la casa (C)

- Valores de N: 10, 150, 400 y 600.
- Valores de M (en función de N): $\frac{N}{2}$

N = 10								
	L_A	**R_A**	**Valor intervalo A**	**L_B**	**R_B**	**Valor intervalo B**	**Mejor opción**	**Tiempo ejecución** (en μs)
M = 5 Casa: 1 A: 8 B: 6	**99,4**	**130,92**	31,52	**31,4**	**39,8**	8,4	**Empresa B**	5982

Tabla 5: pruebas para grafo no conexo con N = 10 y M = 5

N = 150								
	L_A	**R_A**	**Valor intervalo A**	**L_B**	**R_B**	**Valor intervalo B**	**Mejor opción**	**Tiempo ejecución** (en μs)

M = 75 Casa: 1 A: 150 B: 117	**3158,64**	**4166,73**	1008,09	**3007,07**	**4379,58**	1372,51	**No se sabe**	621916

Tabla 6: pruebas para grafo no conexo con N = 150 y M = 75

N = 400								
	L_A	**R_A**	**Valor intervalo A**	**L_B**	**R_B**	**Valor intervalo B**	**Mejor opción**	**Tiempo ejecución** (en μs)
M = 200 Casa: 163 A: 26 B: 31	**56,67**	**94,63**	37,96	**46,19**	**91,9**	45,71	**Empresa B**	272086

Tabla 7: pruebas para grafo no conexo con N = 400 y M = 200

N = 600								
Valores M \ tiempos	**L_A**	**R_A**	**Valor intervalo A**	**L_B**	**R_B**	**Valor intervalo B**	**Mejor opción**	**Tiempo ejecución** (en μs)
M = 300 Casa: A: B:	(*)							

Tabla 8: pruebas para grafo no conexo con N = 600 y M = 300. Debido al alto número de aristas, el algoritmo de formación del grafo en ocasiones intenta dividir por 0 para asignar las probabilidades, por lo que el grafo no es generado correctamente

4.3. Grafos cíclicos

Como se explica en el punto 3.1, estos grafos conforman un polígono de tantos lados como vértices tenga el grafo, Por tanto, el número de vértices y aristas es igual.

Valores de N: 10, 150, 400 y 600

Valores de M (en función de N): N

N = 10								
	L_A	**R_A**	**Valor intervalo A**	**L_B**	**R_B**	**Valor intervalo B**	**Mejor opción**	**Tiempo ejecución** (en μs)
M =10 Casa: 10 A: 3 B: 9	**311,765**	**432,56**	120,795	**112,03**	**271,465**	159,435	**Empresa B**	22919

Tabla 9: pruebas para grafo cíclico con N = 10 y M = 10

N = 150								
	L_A	**R_A**	**Valor intervalo A**	**L_B**	**R_B**	**Valor intervalo B**	**Mejor opción**	**Tiempo ejecución** (en μs)
M = 150 Casa: A: B:								

Tabla 10: pruebas para grafo cíclico con N = 150 y M = 150. El algoritmo no es capaz de encontrar una solución para este tipo de grafo a partir de este punto.

4.4. Grafo en estrella

Como también se explica en el punto 3.1, estos grafos conforman una forma de estrella, de forma que N - 1 vértices están unidos únicamente al vértice central, por lo que este está unido a todos los demás.

Valores de N: 10, 150, 400 y 600

Valores de M (en función de N): N - 1

N = 10								
	L_A	**R_A**	**Valor intervalo A**	**L_B**	**R_B**	**Valor intervalo B**	**Mejor opción**	**Tiempo ejecución** (en µs)
M = 9 Casa: 6 A: 1 B: 5	**303,35**	**437,68**	134,33	**399,88**	**555,62**	155,74	**Empresa A**	16933

Tabla 11: pruebas para grafo en estrella con N = 10 y M = 9

N = 150								
	L_A	**R_A**	**Valor intervalo A**	**L_B**	**R_B**	**Valor intervalo B**	**Mejor opción**	**Tiempo ejecución** (en µs)
M = 149 Casa: 143 A: 10 B: 88	**8163,19**	**10809,7**	2646,55	**6771,41**	**9342,65**	2571,24	**Empresa B**	1609572

Tabla 12: pruebas para grafo en estrella con N = 150 y M = 149

N = 400								
	L_A	**R_A**	**Valor intervalo A**	**L_B**	**R_B**	**Valor intervalo B**	**Mejor opción**	**Tiempo ejecución** (en μs)
M = 399 Casa: 396 A: 227 B: 131	**3286,02**	**4899,93**	1613,91	**2787,05**	**3949,39**	1162,34	**Empresa B**	1963556

Tabla 13: pruebas para grafo en estrella con N = 400 y M = 399

N = 600								
	L_A	**R_A**	**Valor intervalo A**	**L_B**	**R_B**	**Valor intervalo B**	**Mejor opción**	**Tiempo ejecución** (en μs)
M = 599 Casa: 86 A: 297 B: 236	(*)							

Tabla 14: pruebas para grafo en estrella con N = 600 y M = 599. Debido al alto número de aristas, el algoritmo de formación del grafo en ocasiones intenta dividir por 0 para asignar las probabilidades, por lo que el grafo no es generado correctamente

En un comienzo se redondeó las probabilidades a dos decimales, lo que en ocasiones hacía que las probabilidades de las carreteras que salían de una intersección no llegasen a sumar 1, por lo que debían ajustarse correctamente. Sin embargo, cuando se comprobó que el número de aristas era considerable (de varios miles como en la mayoría de los casos de prueba), este redondeo y ajuste de las probabilidades hacía que algunas aristas tuvieran un 0.00 de probabilidad para alguno de los dos sentidos (de *A* a *B* o de *B* a *A*).

La causa de esto era que, a mayor número de carreteras, menos probabilidad tendrá cada una de ser escogida por el repartidor, puesto que eran del orden de 0.00001. Al ser estos valores redondeados a dos decimales quedaban en 0.00, anulando así la posibilidad de que el repartidor fuera por una carretera que, en realidad, sí existe. No obstante, se ha decidido no borrar el código implementado para este ajuste y redondeo de probabilidades. Las funciones correspondientes a esta implementación se encuentran descritas en el Anexo II.

5. CONCLUSIONES SOBRE LOS RESULTADOS

Es importante remarcar que las tablas muestran los resultados de experimentos con grafos muy concretos, muchos de ellos generados de forma aleatoria no solo en cuanto a las probabilidades, sino también en cuanto a la forma del grafo (como es el caso de los puntos *4.1* y *4.2*).

Por otro lado, el hecho de que se utilice un algoritmo de tipo probabilista deja muchas decisiones al azar del "repartidor", que se guía por las probabilidades presentes en cada intersección. Los casos de pruebas utilizados se han considerado relevantes y especiales a la hora de ver cómo se comporta el algoritmo implementado, y se puede observar cómo, de manera evidente, el tiempo tiende a crecer a medida que aparecen más carreteras en el mapa.

6. ANEXO

6.1. Anexo I - Función *probabilistic_algorithm* ubicada en el *main.cc*

```
double probabilistic_algorithm(vector<vector<double>> mapa_probabilidades, vector<vector<double>> mapa_tiempos,
                               int n_vertices, int casa, int almacen_empresa) {
    if(casa == almacen_empresa) {
        return 0;
    } else {
        int interseccion_actual = almacen_empresa;
        bool entregado = false;
        vector<int> camino;
        int carreteras_recorridas = 0;
        camino.insert(camino.begin() + 0, interseccion_actual);
        while (!entregado) {
            vector<double> destinos_interseccion;
            for (int j = 0; j < n_vertices; j++) {
                destinos_interseccion.insert(destinos_interseccion.begin() + j, mapa_probabilidades[interseccion_actual - 1][j]);
            }
            interseccion_actual = getRandomNonZeroIndex(destinos_interseccion, n_vertices);
            carreteras_recorridas++;
            if (interseccion_actual == casa) {
                entregado = true;
            }
            camino.insert(camino.begin() + carreteras_recorridas, interseccion_actual);
        }
        cout << "Carreteras recorridas: " << carreteras_recorridas << endl;

        double tiempo_camino = 0;
        for (int j = 0; j < carreteras_recorridas; j++) {
            tiempo_camino += mapa_tiempos[camino[j + 1] - 1][camino[j] - 1];
        }
        cout << "Tiempo total: " << tiempo_camino << endl;

        return tiempo_camino;
    }
}
```

Figura 11: fragmento de código con el algoritmo probabilista

6.2. Anexo II - Explicación de las funciones implementadas, pero no utilizadas, para el redondeo de las probabilidades

- *adjust_probabilities(edge_probabilities, n_vertices)*: ajusta las probabilidades asociadas a cada arista para que sumen 1.
- *calculate_difference_from_1(probabilities)*: calcula la diferencia entre la suma de las probabilidades y 1.