DOI: 10.24275/uama.6749.10172

Posgrado en Optimización

CBI DIVISIÓN DE CIENCIAS BÁSICAS E INGENIERÍA
UAM - Azcapotzalco

Universidad Autónoma Metropolitana
Casa abierta al tiempo Azcapotzalco

Monitoreo con drones en gráficas con viento dinámico

Tesis presentada a la
Universidad Autónoma Metropolitana Azcapotzalco
como requerimiento parcial para obtener el grado de
Maestro en Optimización
por

Ing. Jovanni Manuel López Elisea

Asesores:
Dr. Francisco Javier Zaragoza Martínez
Departamento de Sistemas, UAM Azcapotzalco

Dr. Rodrigo Alexander Castro Campos
Departamento de Sistemas, UAM Azcapotzalco

Ciudad de México, México
Enero de 2024

Dedicatoria

Dedico esta tesis a mí mismo, por haber perseverado y dedicado tanto tiempo y esfuerzo en este proyecto de investigación. A mis padres, quienes me han brindado su amor y apoyo incondicional durante toda mi vida académica. Sin ellos, este logro no hubiera sido posible. A mis asesores, por su sabiduría, guía y motivación en la realización de este trabajo.

Agradecimientos

Quiero agradecer al Dr. Rodrigo Alexander y al Dr. Francisco Javier, por su invaluable apoyo, orientación y sabiduría que me permitieron enriquecer este trabajo con su experiencia y conocimiento. A mis padres y amigos, por su amor, paciencia y constante motivación. A la Universidad Autónoma Metropolitana (UAM), por su generosidad y compromiso en mi formación profesional. Finalmente, deseo expresar mi más profundo agradecimiento a todas aquellas personas que, de una u otra manera, contribuyeron a la realización de este proyecto.

Resumen

Dada una gráfica completa no dirigida, se desea recorrer un subconjunto de sus aristas usando una flotilla de drones. Los drones tienen baterías limitadas que pueden recargarse al regresar a la base y, en principio, el tiempo para recorrer una arista está en función de la distancia entre sus vértices. Sin embargo, ante la presencia de viento el tiempo de recorrer una arista puede depender del sentido en el que se haga. La dificultad del problema aumenta si además la intensidad del viento puede variar de un instante a otro.

En esta tesis se aborda el problema anteriormente descrito para el caso particular en el que los vértices son puntos en el plano, el impacto del viento en los tiempos de recorrido de las aristas está relativamente acotado y el subconjunto de las aristas a recorrer inducen un árbol que abarca todos los vértices excepto la base de los drones. Dado que los drones operan simultáneamente y pueden recorrer distintas partes de la gráfica de manera independiente, se desea minimizar el tiempo que emplea el dron con el recorrido más tardado.

Esta tesis presenta un modelo matemático para resolver el problema de manera exacta, así como tres heurísticas diferentes para obtener buenas soluciones factibles. La primera de estas heurísticas transforma una solución sin viento y sin batería en una solución con viento y batería. La segunda heurística es un algoritmo glotón sin comunicación entre los drones y la última heurística también es un algoritmo glotón, pero con comunicación entre los drones. Aunque el problema abordado resulta ser lo suficientemente difícil como para que su resolución exacta sea inviable en la práctica, las heurísticas diseñadas son fáciles de implementar y obtuvieron resultados razonables en un tiempo corto de cómputo.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

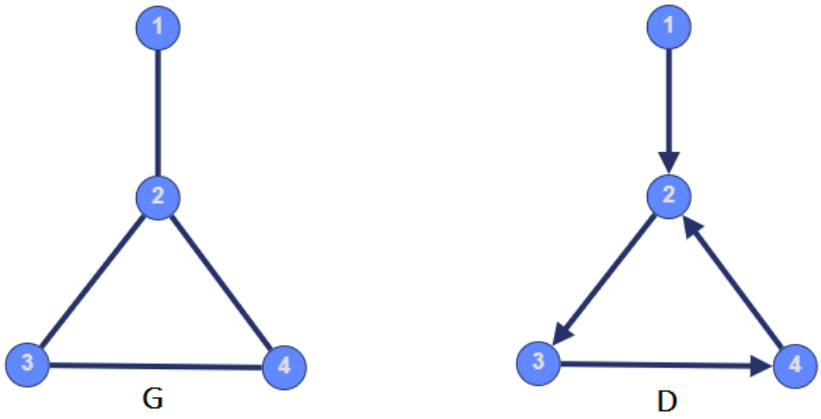
En este capítulo se presentan los conceptos y definiciones necesarias para entender el problema planteado en esta tesis. La mayoría de ellos están relacionados con la teoría de gráficas, aunque también se define qué es un dron y cuáles son sus principales ventajas y limitaciones. El capítulo termina con la presentación del problema general de ruteo y sus diferentes variantes en la literatura.

1.1. Conceptos de teoría de gráficas

Una gráfica G es una pareja $G = (V, E)$ donde V es un conjunto de vértices y E es un conjunto de aristas o pares no ordenados de vértices. Una gráfica dirigida D es una pareja $D = (V, A)$ donde V es un conjunto de vértices y A es un conjunto de arcos o pares ordenados de vértices. Una gráfica mixta M es una tercia $M = (V, E, A)$ donde V es un conjunto de vértices, E es un conjunto de aristas y A es un conjunto de arcos. Conceptualmente, las aristas son conexiones bidireccionales entre vértices, mientras que los arcos son conexiones unidireccionales. La figura 1.1 muestra las representaciones visuales de dos ejemplos de gráficas definidas por:

$$
\begin{aligned}
&G = (V, E) && D = (V, A)\\
&V = V(G) = \{1, 2, 3, 4\} && V = V(D) = \{1, 2, 3, 4\}\\
&E = E(G) = \{\{1, 2\}, \{2, 3\}, \{2, 4\}, \{3, 4\}\} && A = A(D) = \{(1, 2), (2, 3), (3, 4), (4, 2)
\end{aligned}
$$

Dada una gráfica $G = (V, E)$, ésta también puede representarse mediante su gráfica dirigida asociada $\vec{G} = (V, A)$ donde cada arista $e = \{u, v\}$ de E se reemplaza con los dos arcos $e^+ = (u, v)$ y $e^- = (v, u)$ en A. La figura 1.2 muestra la gráfica dirigida $\vec{G}$ asociada a G.



(a) Gráfica (b) Gráfica dirigida

Figura 1.1: Representación visual de dos tipos de gráficas con vértices etiquetados.

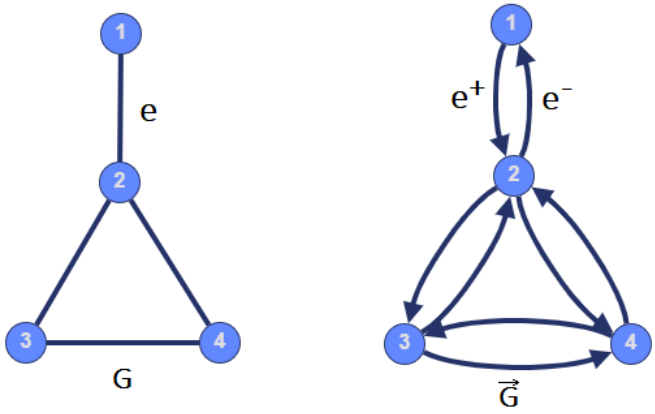


Figura 1.2: Gráfica G y su gráfica dirigida asociada $\vec{G}$.

En una gráfica $G = (V, E)$, $E(v)$ es el conjunto de aristas que inciden en el vértice $v \in V(G)$. De forma similar, en una gráfica dirigida $D = (V, A)$, $A^+(v)$ es el conjunto de arcos que salen del vértice v y $A^-(v)$ es el conjunto de arcos que entran al vértice v.

Un camino en una gráfica G es una secuencia finita $v_0 \to \{v_0, v_1\} \to v_1 \to \{v_1, v_2\} \to v_2 \to \ldots \to \{v_{l-1}, v_l\} \to v_l$ en la que aparecen alternadamente vértices y aristas de G, tal que cada arista tiene por extremos los vértices inmediatamente precedente y siguiente en la secuencia. Por ello, el camino también puede representarse como $v_0 \to v_1 \to v_2 \to \ldots \to v_{l-1} \to v_l$. La longitud de un camino es la cantidad l de aristas que contiene. Un camino es cerrado si $v_0 = v_l$. Se dice que un camino recorre todos los vértices y aristas que lo conforman. Las

definiciones se extienden de forma similar para caminos en gráficas dirigidas y mixtas.

Una gráfica es conexa si hay al menos un camino entre cada pareja de vértices. Se dice que dos vértices pertenecen a diferentes componentes conexas de la gráfica cuando no hay un camino entre ellos. En el caso de gráficas dirigidas, se dice que ésta es fuertemente conexa si hay caminos en ambas direcciones entre cada pareja de vértices.

Es posible asociar costos $c(e)$ a las aristas $e \in E$ de una gráfica. Del mismo modo, es posible asociar costos $c(a)$ a los arcos $a \in A$ de una gráfica dirigida. En ambos casos, se dice que las gráficas tienen pesos en las aristas (o arcos). Cuando dos arcos (u, v) y (v, u) tienen costos $c(u, v) \neq c(v, u)$, se dice que la gráfica dirigida tiene costos asimétricos o con viento.

1.2. Vehículos aéreos no tripulados (drones)

Un vehículo aéreo no tripulado, llamado comúnmente dron, es una aeronave capaz de volar sin la intervención de un humano a bordo. Históricamente, los drones eran aviones pilotados de forma remota, pero cada vez es más común el control autónomo de éstos. Desde sus orígenes militares, los drones han tenido la finalidad de privilegiar la seguridad del ser humano por encima de la seguridad de las propias aeronaves.

Los drones se enfrentan a diversas dificultades como la estabilidad al volar, interferencias electromagnéticas, peligros de colisiones con obstáculos, regulaciones estatales y federales, costo de fabricación y mantenimiento, una capacidad de carga limitada, una autonomía limitada (batería), desafíos de navegación autónoma (condiciones adversas del clima), entre otras. En esta tesis se considerarán únicamente las dos últimas dificultades en el problema a tratar. En cuanto a la autonomía limitada, los drones funcionan usando una batería, la cual tiene una carga limitada y, por lo tanto, debe ser una de las principales consideraciones antes de realizar un vuelo. En cuanto a los desafíos de navegación autónoma, una de las dificultades de volar en condiciones adversas de viento es que se requiere consumir una mayor cantidad de batería si el dron debe volar con el viento en contra. En un uso excesivo de potencia, los motores y la batería pueden sobrecalentarse y sufrir daños irreparables.

1.3. El problema general de ruteo

El problema general de ruteo se define de la siguiente manera. Sea una gráfica mixta fuertemente conexa $M = (V, E, A)$ que consta de un conjunto V de vértices, un conjunto E de aristas y un conjunto A de arcos. Dados subconjuntos específicos $V' \subseteq V, E' \subseteq E, A' \subseteq A$ y una función de pesos no negativos sobre las aristas y los arcos, se desea encontrar un camino

cerrado de peso mínimo que recorra obligatoriamente V', E' y A'. El problema general de ruteo tiene muchas aplicaciones en la vida real, como la planificación de rutas de transporte, la optimización de rutas de entrega, la planificación de rutas de recolección de basura, la planificación de rutas de servicios de emergencia, entre otros. Por lo tanto, es un problema importante en la investigación de operaciones y la optimización combinatoria.

En un estudio del problema general de ruteo se listan distintos casos especiales que dependen de qué vértices o qué aristas o arcos son obligatorios de recorrer [11]. De ahí surgen distintas variantes del mismo problema, por ejemplo:

- Problema del agente viajero (todos los vértices son obligatorios, ninguna arista lo es).
- Problema del cartero chino (todos los vértices y aristas son obligatorios).
- Problema del cartero rural (algunas aristas son obligatorias, ningún vértice lo es).

La tabla 1.1 lista las variantes más conocidas en la literatura y su complejidad computacional.

Nombre	V'	E'	A'	Complejidad
Problema del agente viajero	V	$\emptyset$		NP-Duro
Problema del agente viajero dirigido	V		$\emptyset$	NP-Duro
Problema del cartero chino	V	E		$O(\|V\|^3)$
Problema del cartero chino dirigido	V		A	$O(\|V\|^3 \log_2(\|A\|))$
Problema del cartero chino mixto	V	E	A	NP-Duro
Problema del cartero rural	$\emptyset$	$\subseteq E$		NP-Duro
Problema del cartero rural dirigido	$\emptyset$		$\subseteq A$	NP-Duro
Problema del cartero rural mixto	$\emptyset$	$\subseteq E$	$\subseteq A$	NP-Duro

Tabla 1.1: Diferentes variantes del problema general de ruteo

Que un problema sea NP-Duro implica que no se conoce un algoritmo para resolverlo que sea polinomial con respecto al tamaño de la entrada. Por lo tanto, normalmente se usan algoritmos heurísticos que proporcionan soluciones razonables en un tiempo corto.

Capítulo 2

Antecedentes

En este capítulo se presenta brevemente el problema del cartero rural y el problema de ruteo de vehículos, los cuales están directamente relacionados con el problema a tratar en esta tesis. Para el primero se resalta su versión con viento, mientras que para el segundo se resalta su versión con múltiples vehículos de uso simultáneo.

La literatura presenta modelos matemáticos, estudio de poliedros y técnicas de resolución tanto exactas como heurísticas. Aunque existen casos específicos para los que los problemas se pueden resolver en tiempo polinomial, la mayoría de las versiones de estos problemas son NP-Duras. La mayor parte del trabajo más reciente se concentra en proponer heurísticas.

2.1. El problema del cartero rural

El problema del cartero rural es un problema de optimización que se refiere a la planificación de rutas para un cartero que debe entregar correo a un conjunto de clientes ubicados en una zona rural. El objetivo es minimizar la distancia total recorrida por el cartero, considerando que los clientes a atender se encuentran ubicados en lo que correspondería con un subconjunto de las aristas de una gráfica. Este problema es importante en la logística y el transporte, ya que su estudio ha ayudado a reducir los costos y mejorar la eficiencia de la entrega de correo. A pesar de ser un problema NP-Duro en general, se sabe que el problema puede resolverse en tiempo polinomial si las aristas obligatorias a recorrer inducen una gráfica conexa [12].

El problema del cartero rural se presenta a profundidad en un estudio disponible en [7], el cual fue publicado en 1995 por H. A. Eiselt, M. Gendreau y G. Laporte en la revista Operations Research. Se proporciona una visión general completa del problema del cartero rural y su importancia en la logística y el transporte. Además, se presentan varios enfoques para resolver el problema, incluyendo modelos matemáticos a resolver mediante el algoritmo

de ramificación y corte, además de algunas heurísticas. Los autores también discuten algunas extensiones del problema, como las que ocurren al incluir ventanas de tiempo. Se presentan algunos resultados computacionales y se comparan los diferentes enfoques.

En [6] se introduce la versión con viento del problema y se estudian a profundidad las familias de desigualdades de su poliedro. Esto les permite a los mismos autores presentar en [5] un avanzado algoritmo de generación de ramificación y corte implementado para ejecutarse bajo el solucionador CPLEX. En [1] se presentan tres heurísticas que hacen uso de una extensión euclidiana de la gráfica inducida por las aristas a recorrer, de modo que se aproveche el hecho de que el problema del cartero chino con viento se puede resolver en tiempo polinomial. En [13] se estudia una versión del problema en donde, dependiendo de condiciones periódicas de tiempo, algunas aristas pueden recorrerse sólo una vez, o bien, necesitan recorrerse dos veces y en sentidos opuestos. Se presenta un sofisticado modelo de programación matemática y se resuelven instancias de cuarenta vértices parametrizadas por la proporción de aristas afectadas por el tiempo.

2.2. El problema del ruteo de vehículos

El problema del ruteo de vehículos (VRP por las siglas en inglés de *Vehicle Routing Problem*) considera al problema general de ruteo como un caso especial que usa un único vehículo para recorrer la gráfica. En contraste, el problema del ruteo de vehículos puede emplear múltiples vehículos que pueden recorrer la gráfica de manera secuencial o al mismo tiempo. Naturalmente, es posible estudiar la situación en la que existe una flotilla de m vehículos que pueden operar de manera independiente y simultánea. Normalmente se busca minimizar la suma de los costos de las rutas que recorrieron los m vehículos, o bien, minimizar el costo de la ruta más larga. El problema del cartero chino con m vehículos se puede resolver en tiempo polinomial si se busca minimizar la suma de los costos, pero es NP-Duro si la gráfica tiene viento o si se busca minimizar la ruta más larga [14, 8].

El artículo [2] se enfoca en el problema del cartero rural con viento y múltiples vehículos. El objetivo es encontrar un conjunto de rutas para los vehículos que cubran todos los arcos requeridos y minimicen la longitud de la ruta más larga. Para resolver este problema, se utilizan diferentes técnicas, como la formulación de programación lineal entera mixta y la generación perezosa de desigualdades y de cortes. Se discuten diferentes tipos de cortes, como las de conectividad, paridad y corte R-impar donde R es la cantidad de componentes conexas inducidas por las aristas a recorrer. Los autores aplican las ideas previas en la implementación de un algoritmo de ramificación y corte, para después mostrar resultados computacionales.

En general, los resultados son alentadores, ya que el algoritmo es capaz de resolver la mayoría de las instancias de pruebas en un tiempo razonable. Sin embargo, se señala que aún hay margen para mejorar la eficiencia y la escalabilidad.

En [3] se presenta una metaheurística para el problema del cartero rural con viento y múltiples vehículos para minimizar la ruta más larga. La metaheurística está basada en búsqueda local con multiarranque. Una parte importante del algoritmo es agregar algunas aristas al conjunto de aristas requeridas para asegurarse que la gráfica inducida por éstas sea conexa. La metaheurística emplea una vecindad variable durante su proceso de iteración y alcanza valores cercanos al óptimo para dos o tres vehículos, empeorando muy lentamente conforme el número de vehículos aumenta.

Finalmente, en [4] se presenta un estudio de los problemas de ruteo cuando los vehículos son drones. Se menciona que un dron normalmente puede moverse en el plano y no necesita seguir la red formada por las aristas de la gráfica, además de que podría ocurrir que un dron dejara de recorrer una arista a medio camino. El artículo menciona como trabajo futuro estudiar el efecto de recargar la batería de los drones, así como el impacto del viento.

Capítulo 3

Monitoreo con drones en gráficas con viento dinámico

Informalmente, el problema a tratar en esta tesis es una variante del problema del cartero rural con viento, donde este último es dinámico o dependiente del tiempo y donde además se usa una flotilla de drones para recorrer las aristas requeridas de la gráfica. En este capítulo se describe de manera formal el problema y las capacidades que tendrán los drones para recorrer la gráfica. También se presenta un modelo matemático para el problema.

3.1. Descripción del problema

Sea $G = (V, E)$ una gráfica completa en el plano y sea $d_{i,j}$ la distancia euclidiana entre los vértices i y j. Dado un instante k en el tiempo, se define $c_{i,j,k} \in \mathbb{R}$ como el tiempo que tarda en recorrerse la arista de i a j (en ese sentido) cuando comienza a recorrerse en el instante k. Se tiene que $c_{i,j,k} = \alpha_{i,j,k} d_{i,j}$ donde $\alpha_{i,j,k} \in [\frac{1}{2}, 2]$ es un factor aleatorio de viento. De manera especial se define $c_{i,i,k} = 1$ para permitir que un dron espere una unidad de tiempo en el vértice i. Sea $F \subset E$ el subconjunto de aristas de G que necesitan recorrerse, tal que la gráfica inducida R es un árbol que abarca todos los vértices de G excepto el vértice base 0.

Se desean calcular m caminos cerrados que partan simultáneamente de la base en el instante $k = 0$. El entero m corresponde con la cantidad de drones disponibles, los cuales cuentan con baterías de duración b que se recargan de manera instantánea en la base. Por esta razón, también se pide que ningún camino contenga un subcamino de costo mayor a b que no incluya a la base. Toda arista de F debe recorrerse en al menos uno de los caminos. Se desea minimizar el máximo del tiempo incurrido por cada camino.

Cuando no se imponen restricciones sobre la gráfica inducida R por F, el problema con un

dron y viento estático es equivalente al problema del cartero rural, el cual es NP-Duro [10]. Cuando R inducida es conexa, pero hay más de un vehículo disponible y se desea minimizar el costo del camino más caro, el problema es NP-Duro en gráficas con costos generales [14, 8]. Se desconoce la complejidad del problema cuando es simultáneamente cierto que R es un árbol y que la gráfica es completa con costos euclidianos, los cuales a su vez son afectados por viento variable en el tiempo pero acotado por una constante. Sin embargo, considérese la partición de un árbol T en m componentes conexas disjuntas en aristas $T_1, T_2, \ldots, T_m$, tales que $E(\bigcup T_i) = E(T)$. Si se desea minimizar el número de aristas de la componente conexa con más aristas, el problema es NP-Duro [15]. Por esta razón, es altamente probable que el problema tratado en esta tesis también sea NP-Duro.

3.2. Modelo de programación lineal entera

Dado que un modelo de programación lineal entera necesitará una variable de decisión para cada instante de tiempo que tenga sentido considerarse, es importante suponer que la instancia a resolver es factible. Esto significa que la duración b de la batería debe ser suficiente como para que un único dron se dirija a cada arista requerida, la recorra y después regrese a la base a recargar, todo con viento en contra. Definiremos $k_{\text{máx}}$ como la máxima cantidad de unidades de tiempo que se necesitan para resolver una instancia bajo estas circunstancias.

A continuación, se presenta el modelo de programación completo y se explica a detalle inmediatamente después. El modelo funciona con costos $c_{i,j,k}$ no negativos y con $F \subset E$.

V	: conjunto de vértices de G ($V = \{0, 1, 2, \ldots, \lvert V\rvert - 1\}$).
E	: aristas de G.
F	: aristas requeridas que abarcan $V \setminus \{0\}$.
H	: conjunto de drones.
K	: conjunto de instantes de tiempo a considerar ($K = \{0, 1, 2, \ldots, k_{\text{máx}} - 1\}$).
M	: (constante) valor positivo grande.
b	: (constante) duración de la batería del dron.
$c_{i,j,k}$	: (constante) tiempo que toma recorrer la arista de i a j si un dron sale de i en el instante k.
$k_{\text{máx}}$	: (constante) Máxima cantidad de unidades de tiempo que se necesitan considerar.
$x^h_{i,j,k}$	: (variable binaria) ¿el dron h partirá de i rumbo a j en el instante k?
e^h_k	: (variable entera no negativa) nivel de batería que tiene el dron h en el instante k.
t^h	: (variable entera no negativa) tiempo que tardó el dron h en hacer su recorrido.

Tabla 3.1: Conjuntos, constantes y variables del modelo.

$$\text{minimizar} \quad \max\{t^h : h \in H\} \tag{3.1}$$

$$\text{minimizar} \quad \sum_{h\in H} t^h \tag{3.2}$$

sujeto a:

$$\sum_{h\in H}\sum_{k\in K} x^h_{i,j,k} + \sum_{h\in H}\sum_{k\in K} x^h_{j,i,k} \geq 1 \qquad \forall\, \{i,j\} \in F \tag{3.3}$$

$$\sum_{i\in V}\sum_{j\in V} x^h_{i,j,k} \leq 1 \qquad \forall\, k \in K, h \in H \tag{3.4}$$

$$\sum_{j\in V} x^h_{0,j,0} \geq \sum_{i\in V}\sum_{j\in V} x^h_{i,j,k} \qquad \forall\, k \in K, h \in H \tag{3.5}$$

$$\sum_{j\in V}\sum_{k\in K} x^h_{0,j,k} = \sum_{j\in V}\sum_{k\in K} x^h_{j,0,k} \qquad \forall\, h \in H \tag{3.6}$$

$$e^h_0 = b \qquad \forall\, h \in H \tag{3.7}$$

$$e^h_k \geq \sum_{i\in V}\sum_{j\in V} c_{i,j,k} x^h_{i,j,k} \qquad \forall\, k \in K, h \in H \tag{3.8}$$

$$x^h_{i,j,k} \Rightarrow \sum_{i'\in V}\sum_{j'\in V}\sum_{k'=k+1}^{\min(k_{\max},\, k+c_{i,j,k}-1)} x^h_{i',j',k'} = 0 \qquad \forall\, k \in K, h \in H, \{i,j\} \in E \tag{3.9}$$

$$x^h_{i,j,k} \Rightarrow M \sum_{i'\in V}\sum_{j'\in V} x^h_{i',j',k+c_{i,j,k}} \geq \sum_{i'\in V}\sum_{j'\in V}\sum_{\substack{k'=\\ k+c_{i,j,k}}}^{k_{\max}-1} x^h_{i',j',k'} \qquad \forall\, k \in K, h \in H, \{i,j\} \in E \tag{3.10}$$

$$x^h_{i,j,k} \Rightarrow \sum_{i'\in V\setminus\{j\}}\sum_{j'\in V} x^h_{i',j',k+c_{i,j,k}} = 0 \qquad \forall\, k \in K, h \in H, \{i,j\} \in E \tag{3.11}$$

$$x^h_{i,j,k} \Rightarrow e^h_{k+c_{i,j,k}} = e^h_k - c_{i,j,k} \qquad \forall\, k \in K, h \in H, i \in V, j \in V \setminus \{0\} \tag{3.12}$$

$$x^h_{i,0,k} \Rightarrow e^h_{k+c_{i,0,k}} = b \qquad \forall\, k \in K, h \in H, i \in V \tag{3.13}$$

$$t^h = \sum_{i\in V}\sum_{j\in V}\sum_{k\in K} c_{i,j,k} x^h_{i,j,k} \qquad \forall\, h \in H \tag{3.14}$$

La función objetivo (3.1) minimiza el tiempo total del dron con el recorrido más tardado.

$$\text{minimizar} \qquad \text{máx}\{t^h : h \in H\} \tag{3.1}$$

Pueden existir múltiples soluciones con el mismo valor de la primera función objetivo. Por ejemplo, si los tiempos finales de cinco drones para dos soluciones posibles fueran $(4, 7, 3, 5, 7)$ y $(4, 4, 7, 3, 4)$, el número más grande de ambas es 7, siendo éste el valor calculado por la función objetivo (3.1). Para desempatarlas se usa la segunda función objetivo (3.2) que minimiza la suma de los tiempos. Para la primera solución $(4, 7, 3, 5, 7)$ la suma es 26 y para la segunda solución $(4, 4, 7, 3, 4)$ la suma es 22. Por lo tanto, la segunda solución es mejor que la primera al gastar menos batería en total.

$$\text{minimizar} \qquad \sum_h t^h \tag{3.2}$$

La restricción (3.3) indica que cada arista requerida debe recorrerse en al menos un sentido. La restricción (3.4) indica que cada dron sólo puede realizar una acción cada instante.

$$\sum_{h\in H}\sum_{k\in K} x^h_{i,j,k} + \sum_{h\in H}\sum_{k\in K} x^h_{j,i,k} \geq 1 \quad \forall\, \{i,j\} \in F \tag{3.3}$$

$$\sum_{i\in V}\sum_{j\in V} x^h_{i,j,k} \leq 1 \quad \forall\, k \in K, h \in H \tag{3.4}$$

La restricción (3.5) indica que el dron debe salir de la base en el tiempo 0 si es que recorrerá alguna arista en cualquier otro instante. Esto se verifica para cada instante k.

$$\sum_{j\in V} x^h_{0,j,0} \geq \sum_{i\in V}\sum_{j\in V} x^h_{i,j,k} \quad \forall\, k \in K, h \in H \tag{3.5}$$

La restricción (3.6) indica que se entra a la base la misma cantidad de veces que se sale de ella. La restricción (3.7) indica que cada dron comienza con toda la batería.

$$\sum_{j\in V}\sum_{k\in K} x^h_{0,j,k} = \sum_{j\in V}\sum_{k\in K} x^h_{j,0,k} \quad \forall\, h \in H \tag{3.6}$$

$$e^h_0 = b \quad \forall\, h \in H \tag{3.7}$$

La restricción (3.8) indica que el dron h debe tener suficiente batería e para recorrer una

arista en el instante en el que lo hará.

$$e_k^h \geq \sum_{i \in V} \sum_{j \in V} c_{i,j,k} x_{i,j,k}^h \quad \forall\, k \in K, h \in H \tag{3.8}$$

La restricción (3.9) indica que, si el dron h va a recorrer una arista, no puede realizar ninguna otra acción mientras la está recorriendo.

$$x_{i,j,k}^h \Rightarrow \sum_{i' \in V} \sum_{j' \in V} \sum_{k'=k+1}^{\text{mín}(k_{\text{máx}},\, k+c_{i,j,k}-1)} x_{i',j',k'}^h = 0 \quad \forall\, k \in K, h \in H, \{i,j\} \in E \tag{3.9}$$

La restricción (3.10) indica que, si un dron recorrerá una arista después de terminar de recorrer la arista actual, entonces debe comenzar a recorrer la siguiente inmediatamente después de terminar la actual.

$$x_{i,j,k}^h \Rightarrow M \sum_{i' \in V} \sum_{j' \in V} x_{i',j',k+c_{i,j,k}}^h \geq \sum_{i' \in V} \sum_{j' \in V} \sum_{\substack{k'= \\ k+c_{i,j,k}}}^{k_{\text{máx}}-1} x_{i',j',k'}^h \quad \forall\, k \in K, h \in H, \{i,j\} \in E \tag{3.10}$$

La restricción (3.11) indica que, cuando un dron llega a un vértice j después de recorrer una arista, el dron no puede salir de un vértice distinto al j en el instante inmediato posterior.

$$x_{i,j,k}^h \Rightarrow \sum_{i' \in V \setminus \{j\}} \sum_{j' \in V} x_{i',j',k+c_{i,j,k}}^h = 0 \quad \forall\, k \in K, h \in H, \{i,j\} \in E \tag{3.11}$$

La restricción (3.12) indica que, si el dron recorre una arista que no lo lleva a la base, entonces la energía que le queda es la energía que tenía menos la que usó para recorrer la arista. En cambio, la restricción (3.13) indica que si el dron recorre una arista que lo lleva a la base, entonces la energía se recarga.

$$x_{i,j,k}^h \Rightarrow e_{k+c_{i,j,k}}^h = e_k^h - c_{i,j,k} \quad \forall\, k \in K, h \in H, i \in V, j \in V \setminus \{0\} \tag{3.12}$$

$$x_{i,0,k}^h \Rightarrow e_{k+c_{i,0,k}}^h = b \quad \forall\, k \in K, h \in H, i \in V \tag{3.13}$$

Para cada dron se calculará la suma de los tiempos incurridos en los viajes que realizó. Para eso se usa la restricción (3.14).

$$t^h = \sum_{i \in V} \sum_{j \in V} \sum_{k \in K} c_{i,j,k} x_{i,j,k}^h \quad \forall\, h \in H \tag{3.14}$$

Capítulo 4

Métodos de resolución y pruebas

En este capítulo se presenta el diseño de un generador de instancias y posteriormente se describe la implementación del modelo matemático mediante la interfaz de programación del solucionador Gurobi [9]. Además, se presentan tres heurísticas que generan soluciones factibles para el problema. Así mismo, se presenta un sistema gráfico que permite visualizar una instancia y la ejecución de una solución. Finalmente se presentan los resultados obtenidos por todas las técnicas de resolución al ejecutarse sobre un conjunto de instancias de prueba.

4.1. Generador de instancias

Como crear instancias para el problema de manera manual es un arduo trabajo, se decidió desarrollar un software generador de instancias programado en C++ que facilitara esta labor. Los pasos que realiza el generador de instancias se describen a continuación:

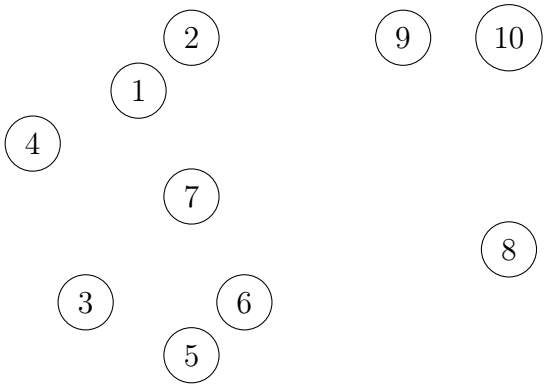
1. El usuario ingresa un número n y el generador de instancias genera n vértices numerados del 1 a n con coordenadas enteras y posiciones aleatorias en el plano.

2. Se usa un algoritmo similar al de Kruskal para elegir un conjunto de aristas que formen un árbol abarcador, sólo que en este caso ignoraremos los pesos de las aristas.

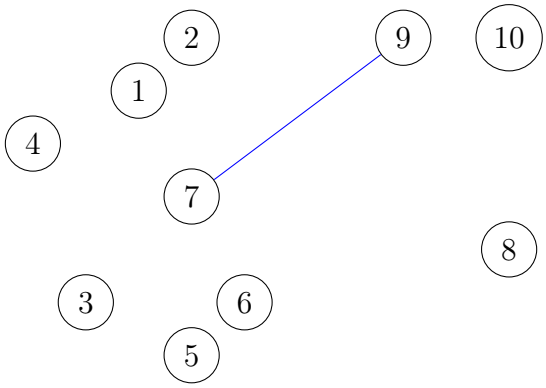
   a) Se genera un arreglo con las $\frac{n(n-1)}{2}$ aristas de la gráfica completa.

   b) Se revuelve el arreglo de las aristas para agregar no determinismo.

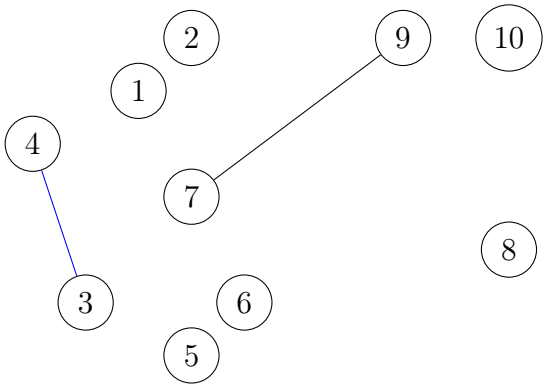
   c) Se recorre el arreglo de aristas, considerando cada una a la vez. Si los extremos de la arista actual no pertenecen a la misma componente conexa de lo que llevamos construido, entonces se agrega para unir componentes conexas. En caso contrario, se descarta la arista.



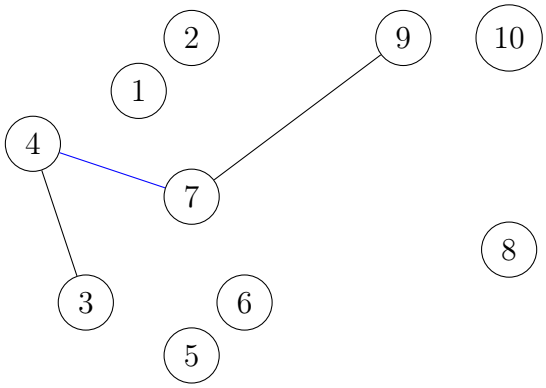
(a) Vértices elegidos para la gráfica completa.



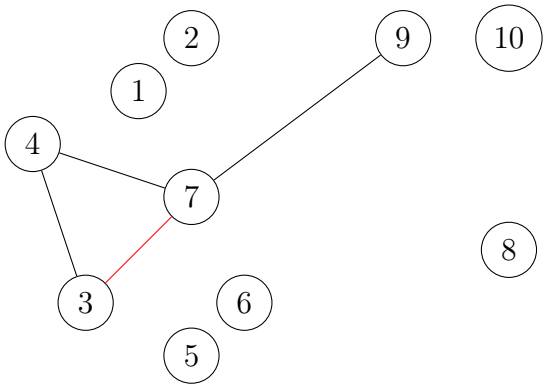
(b) Agregar arista {7,9} entre componentes.



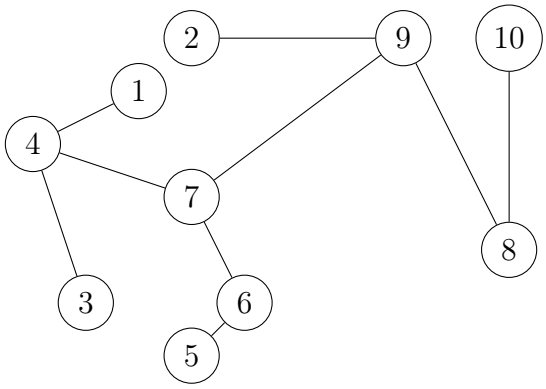
(c) Agregar arista {3,4} entre componentes.



(d) Agregar arista {4,7} entre componentes.



(e) Descartando arista {3,7} entre misma componente.



(f) Ejemplo de árbol abarcador.

Figura 4.1: Ejemplo tras ejecutar los pasos 1 y 2 del generador.

La Figura 4.1 muestra un ejemplo de gráfica con $n = 10$ y un posible árbol abarcador. Las aristas de ese árbol serán las del conjunto F de aristas requeridas.

3. Se genera un vértice adicional etiquetado con el entero 0, el cual será considerado la base de los drones.

4. Se calcula la distancia euclidiana entre cada pareja de vértices para crear la matriz de distancias con los valores d_{ij}, la cual es de tamaño $(n+1) \times (n+1)$.

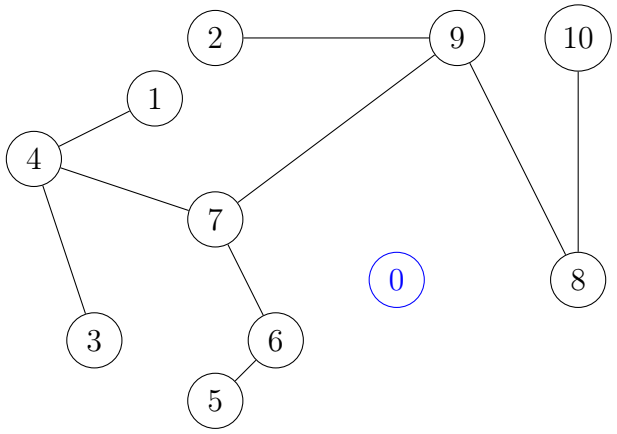


Figura 4.2: Ejemplo de árbol abarcador y el vértice base de la gráfica.

5. En el peor caso, un único dron debe recorrer todas las aristas y éste debe ir a un vértice que es extremo de una arista requerida, recorrer tal arista y regresar a la base a recargar su batería. Se calculan dos valores s y q, donde s es la suma de la longitud del recorrido $(0, i, j, 0)$ (es decir, el valor de $d_{0,i} + d_{i,j} + d_{j,0}$) de todas las aristas requeridas y donde q es el máximo de tales sumandos.

6. Recordemos que los factores de viento aleatorios $\alpha_{i,j,k}$ están acotados en el intervalo $[\frac{1}{2}, 2]$. Se puede afirmar que el tiempo que toma recorrer la gráfica usando un único dron y con el viento siempre en contra es a lo mucho el doble de lo que toma recorrerla sin viento. Por esta razón, la cantidad máxima de unidades de tiempo a considerar es $k_{\text{máx}} = 2s$. Además, para generar siempre una instancia factible, la capacidad de las baterías de los drones necesita ser mayor o igual a la mínima indispensable. Para todas las instancias, la capacidad de las baterías se definirá como $b = 2qn^{0.25}$, el cual es un valor que parece permitir que los drones exploren una parte pequeña pero significativa

de la gráfica antes de necesitar recargar. Nótese que una capacidad de $2q$ es suficiente para recorrer cualquier arista individual en el peor caso, pero quisiéramos baterías ligeramente más grandes que escalen muy poco con el tamaño de la gráfica, ya que la capacidad de almacenamiento de las baterías mejora lentamente en la vida real.

5. Para completar la instancia es necesario generar los costos $c_{i,j,k}$ para todo el horizonte de tiempo. Se generan $2s$ matrices aleatorias mediante el producto de cada $d_{i,j}$ por su factor de viento aleatorio $\alpha_{i,j,k}$, generando así la matriz de costos $c_{i,j,k}$ para $0 \leq k < k_{\text{máx}}$.

4.1.1. Entrada del generador de instancias

Para generar una instancia es necesario indicar el número de vértices n del árbol a recorrer. Esta cantidad no incluye al vértice base, pues no forma parte del árbol.

4.1.2. Salida del generador de instancias

La instancia generada se describe mediante el número de vértices n, la capacidad b de la batería, el tamaño del horizonte de tiempo $k_{\text{máx}}$, las coordenadas de la base, las coordenadas de los n vértices del árbol, la matriz de los costos $c_{i,j,k}$ y $n-1$ parejas de enteros $\{i, j\}$ que denotan los extremos de las aristas del árbol.

4.2. Resolución del modelo de programación con Gurobi

La programación matemática sirve para encontrar la solución óptima de un modelo descrito mediante desigualdades. Gurobi es un tipo especial de software llamado solucionador lineal, el cual permite resolver modelos lineales mixtos (con variables continuas y enteras) [9]. Gurobi no tiene una interfaz gráfica, pero se puede interactuar con él a través de consola o haciendo uso de su interfaz de programación disponible para varios lenguajes de alto nivel como C++, Python, etc. Dada una instancia del problema, se implementó un programa en C++ que construye el modelo correspondiente y después le solicita a Gurobi resolverlo.

El problema de esta tesis es difícil, por lo que no se espera que el modelo de programación entera pueda resolverse en un tiempo razonable más que para instancias muy pequeñas. Por esta razón, el modelo se diseñó principalmente para fines expositivos. A pesar de tal suposición, se decidió comprobar su veracidad e intentar resolverlo con Gurobi, ya que éste es uno de los mejores software solucionadores disponibles en la actualidad.

Con ayuda del generador de instancias, inicialmente se creó una instancia de cinco vértices para la cual Gurobi tardó alrededor de 3 minutos en resolverla. Al probar una instancia

de 12 vértices, Gurobi no fue capaz ni siquiera de construir el modelo, pues al cabo de aproximadamente una hora superó el límite de los 128 GB de memoria disponibles en la computadora usada. Sólo hasta que se disminuyó a 6 el número de vértices fue que Gurobi pudo encontrar una solución factible, aunque no necesariamente óptima; después de esperar optimalidad por alrededor de 12 horas, se decidió detener el proceso. Por tal motivo, se comprueba que el modelo de programación entera no se puede resolver en un tiempo razonable más que para instancias muy pequeñas y es necesario usar técnicas heurísticas.

4.2.1. Entrada del solucionador

El programa que se encarga de construir el modelo de programación de una instancia e intentar resolverlo con Gurobi, toma una entrada con la misma estructura que la salida emitida por el generador de instancias. La cantidad m de drones a emplear se calcula automáticamente redondeando $\sqrt{n}$ al entero más cercano.

4.2.2. Salida del solucionador

Se emite el número m de drones empleados, seguido de la siguiente información para cada dron: número a de aristas recorridas, seguido de a tripletas de enteros k, i, j que denotan los extremos i y j de la arista recorrida (en el sentido hecho) y el instante k en el que se recorrió.

4.3. Heurística 1 - Adaptar la solución de otro problema

Esta heurística primero invierte una cantidad tiempo prestablecida en encontrar una buena solución para la versión simplificada del problema en la que no hay viento ni limitaciones de batería. Este cálculo se lleva a cabo con el solucionador Gurobi, haciendo uso de un modelo matemático mucho más simple que ignora el viento de la gráfica y la capacidad de la batería de los drones. A continuación, se presenta un modelo de lineal entera para esta versión del problema. La idea de este modelo es, para cada dron individual, calcular cuántas veces se recorrerá cada arista. Si requerimos que el número de veces que se llega a un vértice sea el mismo número de veces que se sale de él, se generarán recorridos cerrados. Para garantizar que cada dron tiene un único recorrido que además incluye a la base, se puede emplear una idea de flujo: la base es una fuente de flujo infinito que se envía a través de las aristas que el dron recorrerá; un vértice ordinario por el que el dron pasa debe repartir el flujo que llega a él, reteniendo además una parte positiva de éste.

V	: conjunto de vértices de G ($V = \{0, 1, 2, \ldots, \lvert V\rvert - 1\}$).
E	: aristas de G.
F	: aristas requeridas que abarcan $V \setminus \{0\}$.
H	: conjunto de drones.
M	: (constante) valor positivo grande.
$d_{i,j}$	: (constante) tiempo que toma recorrer la arista de i a j.
$x^h_{i,j}$	: (variable entera) cantidad de veces que el dron h partirá de i rumbo a j
$y^h_{i,j}$	: (variable real) cantidad de flujo que cede el vértice i al j al recorrerla el dron h
t^h	: (variable entera no negativa) tiempo que tardó el dron h en hacer su recorrido.

Tabla 4.1: Conjuntos, constantes y variables del modelo simplificado.

$$\text{minimizar} \qquad \text{máx}\{t^h : h \in H\} \tag{4.1}$$

$$\tag{4.2}$$

sujeto a:

$$\sum_{h\in H} x^h_{i,j} + \sum_{h\in H} x^h_{j,i} \geq 1 \qquad \forall\, \{i,j\} \in F \tag{4.3}$$

$$\sum_{i\in V}\sum_{j\in V} x^h_{i,j} = \sum_{i\in V}\sum_{j\in V} x^h_{j,i} \qquad h \in H \tag{4.4}$$

$$0 \leq y^h_{i,j} \leq M x^h_{i,j} \qquad \forall\, h \in H, i \in V, j \in V \tag{4.5}$$

$$\sum_{j\in V\setminus\{i\}} y^h_{j,i} - \sum_{j\in V\setminus\{i\}} y^h_{i,j} \geq \sum_{j\in V\setminus\{i\}} x^h_{j,i} + \sum_{j\in V\setminus\{i\}} x^h_{i,j} \qquad \forall\, h \in H, i \in V \setminus \{0\} \tag{4.6}$$

$$t^h = \sum_{i\in V}\sum_{j\in V} d_{i,j} x^h_{i,j} \qquad \forall\, h \in H \tag{4.7}$$

Tras intentar resolver el modelo anterior con Gurobi, un programa en C++ toma la mejor solución encontrada y la transforma en una solución factible del problema original con viento y con batería. Esto se hace modificando el recorrido calculado para cada dron usando la siguiente estrategia: si el dron puede recorrer la siguiente arista y luego regresar a la base usando la arista directa, entonces recorre la siguiente arista y se repite la verificación. En caso contrario, el dron regresa a la base usando una arista directa y después regresa al vértice extremo de la siguiente arista a recorrer usando también la arista directa desde la base.

4.3.1. Entrada de la heurística 1

El programa que calcula una solución del problema simplificado toma la información de la instancia, pero ignora la batería y los costos con viento para sólo trabajar con las distancias euclidianas entre los vértices. Un segundo programa, el que adapta la solución calculada en el paso anterior para el problema simplificado, toma la información de la instancia y también la solución calculada en el mismo formato en el que fue emitida.

4.3.2. Salida de la heurística 1

Se emite el costo en tiempo del recorrido más tardado de los drones, seguido de la siguiente información para cada dron: identificador numérico del dron, número de vértices en su recorrido y una secuencia de vértices que denotan dicho recorrido. Estas secuencias siempre comienzan y terminan en el vértice 0.

4.4. Heurística 2 - Algoritmo glotón sin comunicación entre drones

Esta heurística emplea un algoritmo glotón que permite que los drones conozcan la lista de aristas aún pendientes de recorrer cada vez que llegan a un vértice, pero no permite que conozcan las posiciones de los demás drones ni las decisiones que éstos han tomado. La estrategia general es que un dron se dirija al vértice más cercano que sea extremo de una arista requerida aún no recorrida. Las consideraciones para esta heurística son las siguientes:

1. Sea $r = \text{máx}(\{d_{i,0} : i \in v\})$. Cuando a un dron le queda $2r$ o menos de batería, regresa a la base a recargar.

2. Cuando un dron solitario está en el vértice i en el instante k y el vértice tiene aristas pendientes, entonces el dron recorre la arista pendiente con el costo $c_{i,j,k}$ más barato.

3. Cuando un dron solitario está en el vértice i en el instante k y el vértice no tiene aristas pendientes, entonces el dron recorre la arista con el costo $c_{i,j,k}$ más barato que lo lleve a un vértice que sí tenga aristas pendientes.

4. Cuando varios drones llegan al mismo vértice en el mismo instante, se revisan los puntos 2 o 3 para cada uno de forma secuencial.

5. Cuando ya no hay aristas pendientes, los drones regresan a la base.

4.4.1. Entrada de la heurística 2

El programa simplemente toma la información de la instancia. No necesita datos adicionales.

4.4.2. Salida de la heurística 2

El programa emite su solución de la misma manera en que lo hace la heurística 1.

4.5. Heurística 3 - Algoritmo glotón con comunicación entre drones

Esta heurística es similar a la heurística 2, pero permite que los drones conozcan hacia dónde se están dirigiendo los demás drones. En principio, esto permitiría evitar que dos drones se dirijan a recorrer la misma arista. Las consideraciones para esta heurística son las siguientes.

1. Sea $r = \text{máx}(\{d_{i,0} : i \in v\})$. Cuando a un dron le queda $2r$ o menos de batería, regresa a la base a recargar.

2. Cuando un dron solitario está en el vértice i en el instante k y el vértice tiene aristas pendientes, entonces el dron recorre la arista pendiente con el costo $c_{i,j,k}$ más barato, siempre que el vértice tenga más aristas pendientes que drones dirigiéndose a él.

3. Cuando un dron solitario está en el vértice i en el instante k y el vértice no tiene aristas pendientes, entonces el dron recorre la arista con el costo $c_{i,j,k}$ más barato que lo lleve a un vértice que sí tenga aristas pendientes y que tenga más aristas pendientes que drones dirigiéndose a él.

4. Cuando varios drones llegan al mismo vértice en el mismo instante, se revisan los puntos 2 o 3 para cada uno de forma secuencial.

5. Cuando ya no hay aristas pendientes, los drones regresan a la base.

4.5.1. Entrada de la heurística 3

El programa simplemente toma la información de la instancia. No necesita datos adicionales.

4.5.2. Salida de la heurística 3

El programa emite su solución de la misma manera en que lo hacen las heurísticas 1 y 2.

4.6. Sistema gráfico

Para visualizar las instancias y poder interpretar de mejor manera las soluciones calculadas por las heurísticas, se decidió crear un sistema gráfico que fuese capaz de mostrar de manera animada los recorridos de los drones sobre la gráfica. Se diseñó e implementó un sistema web usando las tecnologías HTML, CSS, SVG y JavaScript con las siguientes características:

- El generador de instancias se tradujo de C++ a JavaScript y se incorporó directamente en el sistema gráfico.
- Las instancias desplegadas por el sistema se pueden exportar a formato texto siguiendo la misma estructura que las que son emitidas por el generador.
- El usuario puede importar instancias propias dadas en formato de texto.
- Hay un recuadro donde se dibujan los vértices, las aristas requeridas y los drones.
- Se indican las coordenadas de los vértices y los identificadores de los extremos de las aristas requeridas. Estos datos se pueden modificar con la interfaz gráfica del sistema.
- Es posible arrastrar los vértices para modificar su posición.
- Es posible ingresar en formato de texto el recorrido de cada dron.
- Se puede visualizar mediante animación el recorrido de cada dron y las aristas requeridas se colorean de manera distinta dependiendo si ya fueron o no recorridas.
- El sistema valida toda la información ingresada. En caso de ingresar un valor inválido o de modificar un valor de manera incorrecta, el sistema emite una alerta.

Las figuras 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 4.10 y 4.11 muestran el sistema gráfico.

Figura 4.3: Pantalla inicial del sistema gráfico.

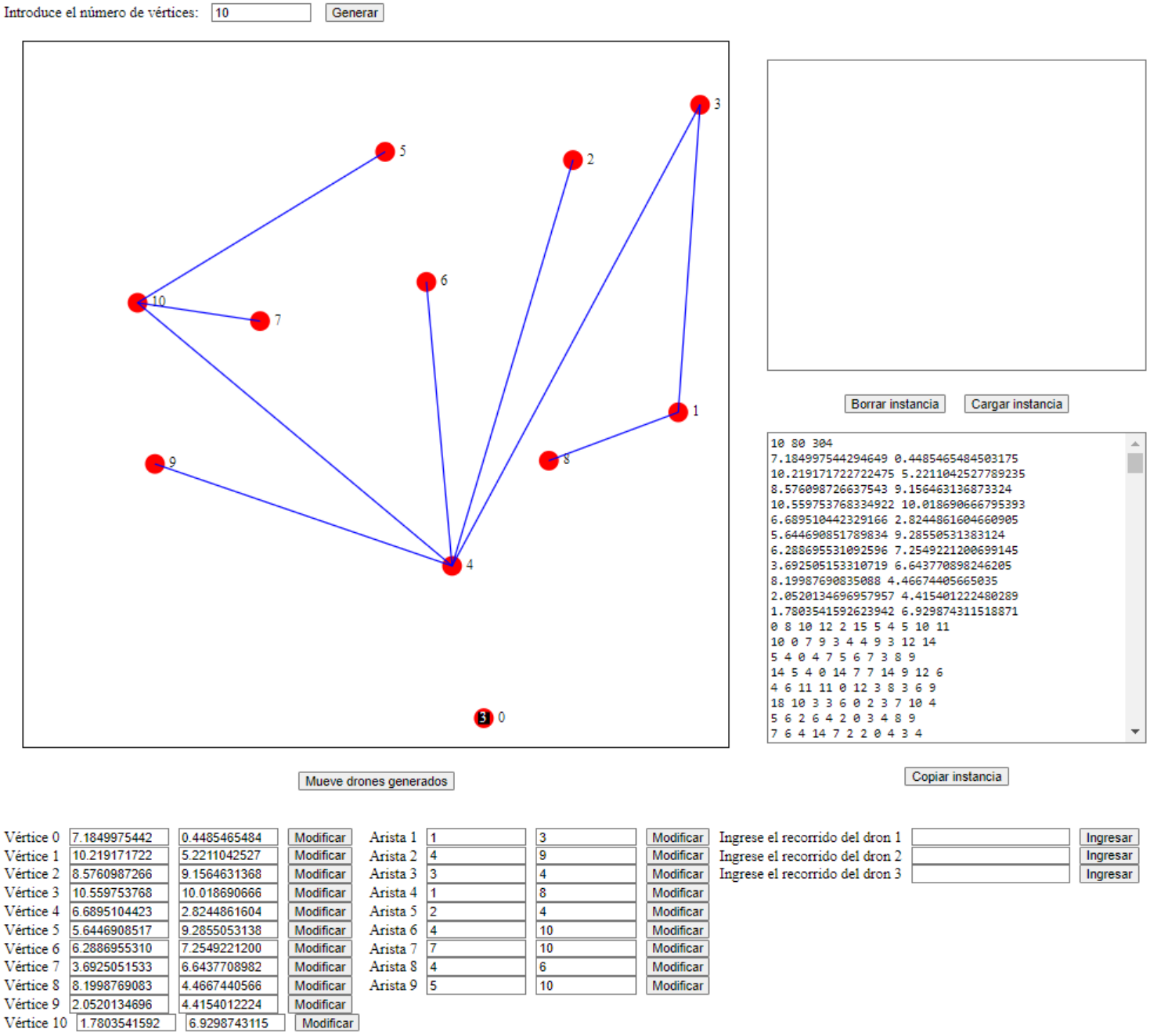


Figura 4.4: Pantalla después de generar una instancia de diez vértices.

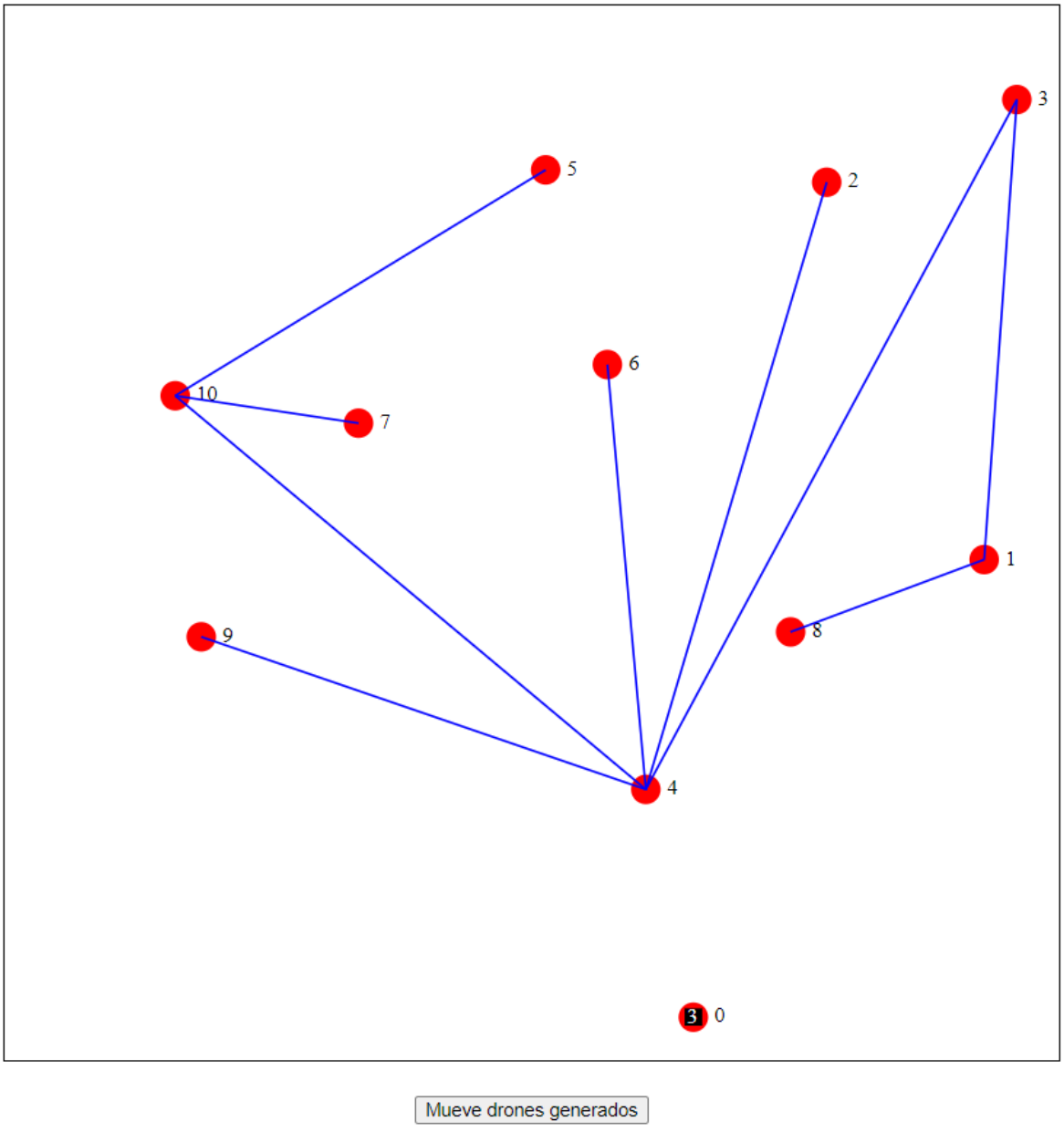


Figura 4.5: Recuadro que muestra los vértices, las aristas, los drones y las animaciones.

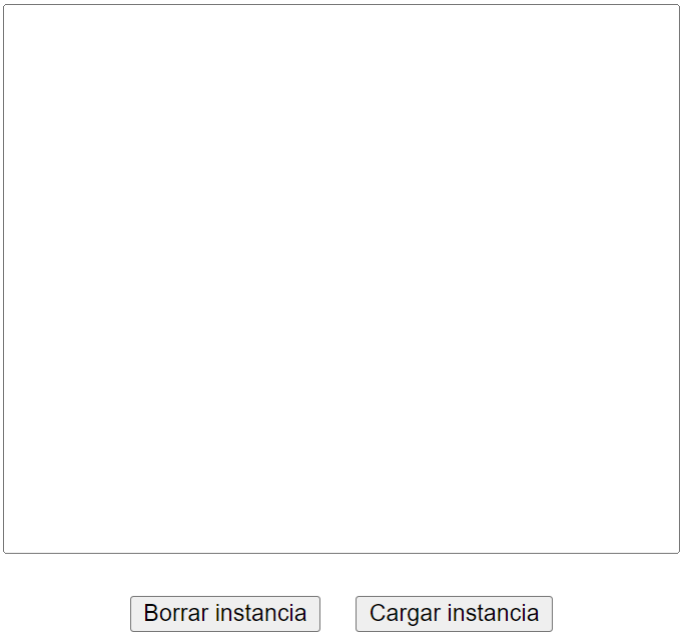


Figura 4.7: Recuadro donde el usuario puede ingresar una instancia.

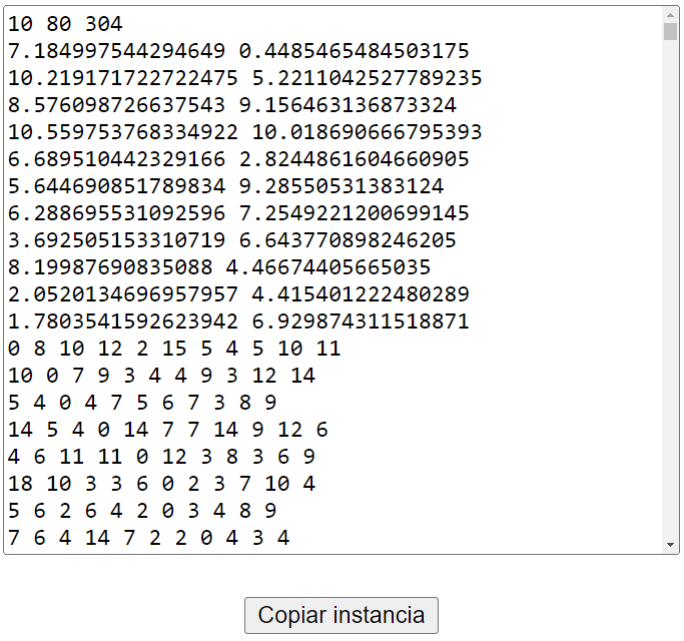


Figura 4.6: Recuadro donde se muestra la instancia generada. Se provee la opción de copiarla.

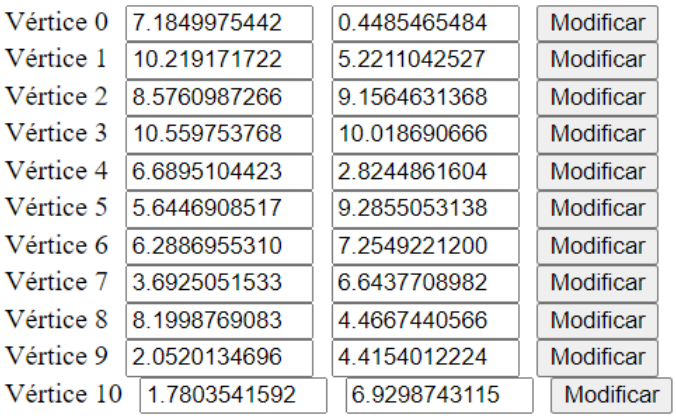


Figura 4.8: Recuadros donde se puede modificar las coordenadas de los vértices.

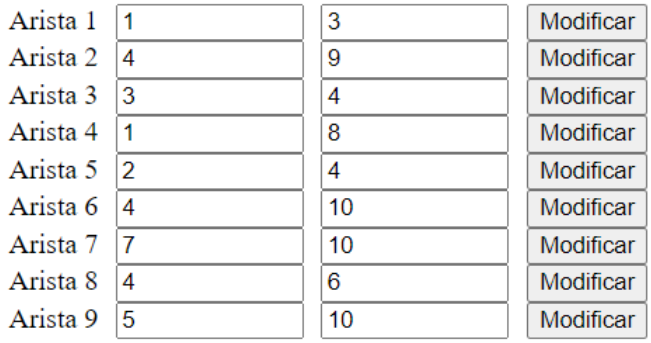


Figura 4.9: Recuadros donde se pueden modificar los vértices de las aristas requeridas.

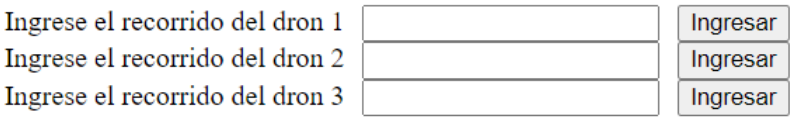


Figura 4.10: Recuadros donde se puede ingresar el recorrido para cada dron.

Figura 4.11: Alerta al ingresar datos incorrectos.

4.7. Pruebas y análisis

Se llevaron a cabo un conjunto de pruebas para observar el comportamiento de las heurísticas. Para esto, se generaron instancias con árboles de desde cinco hasta cincuenta vértices, una por número de vértices, para las cuales se calculó una solución factible usando cada una de las tres heurísticas. Cabe recordar que la heurística 1 – "Adaptar la solución de otro problema" (4.3) requiere como preproceso el resolver un modelo de programación que no considera el viento ni la batería, solución que después se adapta en una solución del problema original.

Para tal preproceso, se ejecutó Gurobi en una computadora multiprocesador con dos CPU AMD Opteron 6174 de 2.2 GHz con 24 núcleos y con 128 GB de RAM. Gurobi encontró la solución óptima del problema simplificado para todas las instancias con 16 o menos vértices en el árbol requerido. Para el resto de las instancias, se usó la mejor solución encontrada después de un tiempo límite de una hora. Las tres heurísticas se ejecutaron en una computadora con un procesador AMD Ryzen 5 4600H y 16 GB de RAM usando un único hilo. En todas las pruebas, la ejecución de las heurísticas no tardó más de una milésima de segundo por instancia. En la tabla 4.2 se muestra el valor alcanzado por el preproceso y por las heurísticas.

Vértices del árbol	Valor alcanzado			
	Preproceso	Heurística 1	Heurística 2	Heurística 3
5	15	**19**	20	20
6	21	**27**	33	33
7	16	**23**	28	28
8	19	**24**	27	26
9	28	43	**30**	**30**
10	25	**31**	34	34
11	38	**54**	64	57
12	35	45	**44**	**44**
13	41	**57**	**57**	**57**
14	42	**59**	64	70
15	43	67	**54**	**54**
16	51	81	**71**	**71**
17	56	73	75	**71**
18	58	**80**	86	**80**
19	68	102	108	**101**
20	84	**122**	126	127
21	64	**87**	109	109

Vértices del árbol	Valor alcanzado			
	Preproceso	Heurística 1	Heurística 2	Heurística 3
22	91	117	**102**	123
23	67	94	**91**	**91**
24	95	**131**	163	153
25	94	134	120	**110**
26	97	**120**	147	147
27	95	138	**121**	**121**
28	112	161	**159**	166
29	107	**147**	157	166
30	112	158	**143**	**143**
31	126	163	**145**	170
32	111	169	176	**168**
33	148	**192**	205	223
34	142	**197**	**197**	218
35	150	218	**174**	235
36	167	233	**203**	225
37	139	**169**	200	186
38	181	261	251	**240**
39	184	277	224	**219**
40	191	265	225	**216**
41	194	291	228	**209**
42	189	284	**234**	**234**
43	189	271	**210**	216
44	196	271	**269**	281
45	212	294	**283**	300
46	266	387	**286**	**286**
47	227	339	**254**	**254**
48	292	421	**325**	328
49	286	393	**349**	**349**
50	234	332	**303**	320

Tabla 4.2: Resultados de las pruebas para las heurísticas. Aparecen subrayados los óptimos del preproceso de la heurística 1. Aparece en negritas el mejor resultado en cada instancia.

En la figura 4.12 se grafican los valores alcanzados para cada instancia con las diferentes

heurísticas. A pesar de que en algunas instancias la heurística 1 alcanzó un mejor valor, en la mayoría de los casos presentó peores resultados en comparación con las otras dos. Además, al tiempo de ejecución de la heurística 1 se le debe sumar el tiempo del preproceso, lo cual la pone en franca desventaja. Para finalizar, se puede observar que la heurística 1 presenta resultados cada vez peores conforme el tamaño de la instancia crece, esto debido a que la solución del preprocesamiento no considera el viento ni la batería y que, al adaptar tal solución para que sí las considere, la cantidad de visitas a la base aumenta drásticamente, empeorando la solución considerablemente.

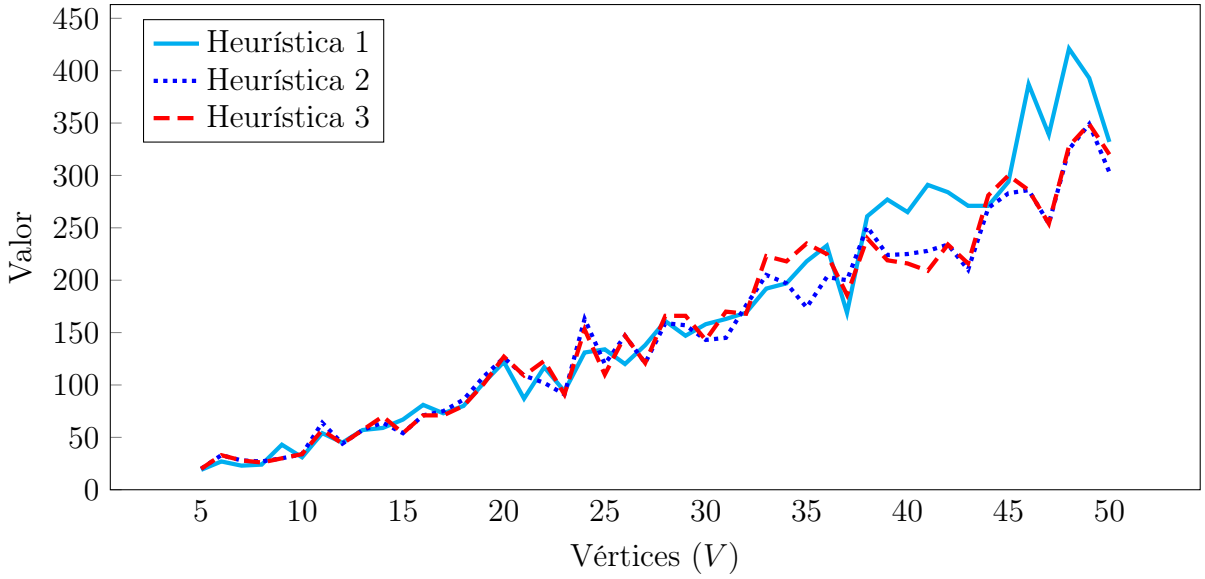


Figura 4.12: Valores alcanzados de las pruebas para las tres heurísticas.

La heurística 2 – "Algoritmo glotón sin comunicación entre drones" (4.4) y la heurística 3 – "Algoritmo glotón con comunicación entre drones" (4.5) presentan casi los mismos resultados con algunas excepciones para situaciones en particular. Esto es esperado debido a que son heurísticas muy similares con la sutil diferencia de que la heurística 3 envía a los vértices sólo a los drones necesarios para recorrer las aristas requeridas de dicho vértice. Es decir, cuando un dron decide recorrer una arista de i a j, éste les avisa a los demás drones que él se dirige a j, por lo tanto, si el vértice j tiene otra arista requerida, ese dron recorrerá esa arista. Esto se hace así para que los demás drones no se dirijan también al vértice j si es que éste sólo tiene una arista requerida que el dron que ya se dirige a él recorrerá. En algunas ocasiones esto puede empeorar la solución en lugar de mejorarla. Por ejemplo, puede ocurrir que el dron llega al vértice j pero ya no tiene batería para recorrer la arista requerida adyacente y

debe regresar a la base; otro dron que estaba cerca pudo haber recorrido tal arista, pero no lo hizo porque confió en que el otro dron lo haría. En situaciones menos desafortunadas, la heurística sí disminuye el trabajo realizado. A pesar de estas diferencias, se puede observar que los valores alcanzados por ambas heurísticas son muy similares entre sí y que en general tienden a ser mejores que los resultados de la heurística 1.

Para finalizar, se desarrolló un algoritmo verificador que comprueba que la salida de las heurísticas es correcta. Esto se hace verificando que se hayan recorrido todas las aristas requeridas y que la batería de los drones nunca se haya agotado.

Capítulo 5

Conclusiones

En esta tesis se abordó el problema de recorrer un subconjunto de las aristas de una gráfica usando una flotilla de drones. Se desea minimizar el tiempo que toma recorrer las aristas requeridas, sujeto a que los drones individuales pueden operar al mismo tiempo, a que tienen baterías limitadas que se recargan al regresar a la base, y a que el tiempo de recorrer una arista se ve afectado por la presencia de viento dinámico. Se cree que el problema es NP-Duro.

Esta tesis presentó un modelo matemático para resolver el problema de manera exacta, así como tres heurísticas diferentes para obtener soluciones factibles. El modelo matemático usa una enorme cantidad de variables que surgen de la necesidad de modelar el viento cambiante en el tiempo. En ese sentido, es desafortunado, pero también esperado, que el solucionador Gurobi no pudiera resolver en un tiempo razonable ni siquiera instancias de una decena de vértices. Por esta razón y debido a la dificultad del problema, el uso de heurísticas es esencial.

La primera de las heurísticas presentadas transforma una solución sin viento y sin restricciones de batería en una solución con viento y batería. La segunda heurística es un algoritmo glotón sin comunicación entre los drones y la última también es un algoritmo glotón, pero con comunicación entre drones. Aunque las soluciones heurísticas pueden no ser óptimas, su ejecución casi instantánea y la aplicabilidad práctica de las soluciones calculadas las convierten en herramientas valiosas en la toma de decisiones.

Se observó que las heurísticas que desde el principio consideran todos los aspectos del problema fueron mejores que la heurística que adaptó o corrigió una solución de una versión más simple del problema. Por otra parte, y en contra de lo esperado, la heurística que permitía que los drones compartieran información entre ellos no siempre generó mejores soluciones que las heurísticas que no lo hacían. Un posible trabajo futuro consiste en planear el recorrido de los drones por etapas, en lugar de tomar únicamente decisiones locales glotonas.

Apéndice A

Código fuente de los algoritmos implementados

A.1. Implementación del modelo de programación entera lineal utilizando la interfaz de programación de C++ de Gurobi.

```
// exacto batería + viento
// siempre se gasta energía, aún al quedarse sobrevolando el
   mismo vértice
// se asume que el tiempo usado == energía gastada
#include <algorithm>
#include <array>
#include <iostream>
#include <math.h>
#include <string>
#include <stdio.h>
#include <gurobi_c++.h>

template<typename... T>
std::string formato(const char* s, const T&... v) {
   char bufer[256 + 1];
   sprintf(bufer, s, v...);
```

```
   return std::string(bufer);
}

struct punto{
    double x, y;
};

int main( ) try{
    // Lectura de la entrada

    int n, b, tiempo;       // número de vértices (sin contar la
       base), batería, tiempo de simulación
    std::cin>> n >> b >> tiempo;

    punto arr[n + 1];       // coordenadas de los puntos (no se
       usarán, pero vendrán en la entrada)
    for (int i = 0; i < n + 1; ++i){
        std::cin >> arr[i].x >> arr[i].y;
    }

    int c[n+1][n+1][tiempo];     // Matrices de cada instante
    for(int k = 0; k < tiempo; ++k){
        for(int i = 0; i < n + 1; ++i){
            for(int j = 0; j < n + 1; ++j){
                std::cin >> c[i][j][k];
                c[i][j][k] = std::max(1, c[i][j][k]);
            }
        }
    }

    int aristas[n-1][2];    // Aristas
    for(int i = 0; i < n - 1; ++i){
        std::cin>> aristas[i][0] >> aristas[i][1];
    }
```

```
int num_drones = lround(sqrt(n));    // por ahora, el número
   de drones no formará parte de la entrada, sino que se
   decidirá de forma independienteg

// Construcción del modelo

GRBEnv ambiente;
GRBModel modelo(ambiente);

GRBVar x[num_drones][n+1][n+1][tiempo];
for(int h = 0; h < num_drones; ++h){
    for(int i = 0; i < n + 1; ++i){
        for(int j = 0; j < n + 1; ++j){
            for(int k = 0; k < tiempo; ++k){
                x[h][i][j][k] = modelo.addVar(0, 1, 0,
                   GRB_BINARY, formato("x%d_%d_%d_%d", h, i,
                   j, k));
            }
        }
    }
}
GRBVar e[num_drones][tiempo];
for(int h = 0; h < num_drones; ++h){
    for(int k = 0; k < tiempo; ++k){
        e[h][k] = modelo.addVar(0, +GRB_INFINITY , 0,
           GRB_INTEGER, formato("e%d_%d", h, k));
    }
}
GRBVar t[num_drones];
for(int h = 0; h < num_drones; ++h){
    t[h] = modelo.addVar(0, +GRB_INFINITY , 0, GRB_INTEGER,
       formato("t%d", h));
}

// Cada arista requerida debe recorrerse al menos una vez en
```

```
   cualquier sentido
for(int a = 0; a < n - 1; ++a){
    int i = aristas[a][0], j = aristas[a][1];
    GRBLinExpr ex;
    for(int h = 0; h < num_drones; ++h){
        for(int k = 0; k < tiempo; ++k){
            ex += x[h][i][j][k];
            ex += x[h][j][i][k];
        }
    }
    modelo.addConstr(ex >= 1);
}

// El dron sólo puede realizar una acción en el tiempo k
for(int h = 0; h < num_drones; ++h){
    for(int k = 0; k < tiempo; ++k){
        GRBLinExpr ex;
        for(int i = 0; i < n + 1; ++i){
            for(int j = 0; j < n + 1; ++j){
                ex += x[h][i][j][k];
            }
        }
        modelo.addConstr(ex <= 1);
    }
}

// Si algún arco se recorrerá en algún momento, entonces es
   necesario salir de la base en el tiempo 0
for(int h = 0; h < num_drones; ++h){
    GRBLinExpr ex1;
    for(int j = 0; j < n + 1; ++j){
        ex1 += x[h][0][j][0];
    }
    for(int k = 0; k < tiempo; ++k){
        GRBLinExpr ex2;
```

```
            for(int i = 0; i < n + 1; ++i){
                for(int j = 0; j < n + 1; ++j){
                    ex2 += x[h][i][j][k];
                }
            }
            modelo.addConstr(ex1 >= ex2);
        }
    }

    // Se entra a la base la misma cantidad de veces que se sale
       de ella
    for(int h = 0; h < num_drones; ++h){
        GRBLinExpr ex1;
        GRBLinExpr ex2;
        for(int j = 0; j < n + 1; ++j){
            for(int k = 0; k < tiempo; ++k){
                ex1 += x[h][0][j][k];
                ex2 += x[h][j][0][k];
            }
        }
        modelo.addConstr(ex1 == ex2);
    }

    // El dron no puede realizar ninguna acción mientras está
       recorriendo una arista
    for(int h = 0; h < num_drones; ++h){
        for(int i = 0; i < n + 1; ++i){
            for(int j = 0; j < n + 1; ++j){
                for(int k = 0; k < tiempo; ++k){
                    GRBLinExpr ex;
                    for(int i2 = 0; i2 < n + 1; ++i2){
                        for(int j2 = 0; j2 < n + 1; ++j2){
                            for(int k2 = k + 1; k2 < std::min(k +
                                c[i][j][k], tiempo); ++k2){
                                ex += x[h][i2][j2][k2];
```

```
                        }
                    }
                }
                modelo.addGenConstrIndicator(x[h][i][j][k],
                   1, ex == 0);
            }
        }
    }
}

// No puede haber tiempos muertos (si un dron recorrerá una
   arista después de la actual, debe comenzar a hacerlo
   inmediatamente)
for(int h = 0; h < num_drones; ++h){
    for(int i = 0; i < n + 1; ++i){
        for(int j = 0; j < n + 1; ++j){
            for(int k = 0; k < tiempo; ++k){
                GRBLinExpr ex1, ex2;
                for(int i2 = 0; i2 < n + 1; ++i2){
                    for(int j2 = 0; j2 < n + 1; ++j2){
                        if (k + c[i][j][k] < tiempo) {
                            ex1 += x[h][i2][j2][k + c[i][j][k
                               ]];
                        }
                    }
                }
                for(int i2 = 0; i2 < n + 1; ++i2){
                    for(int j2 = 0; j2 < n + 1; ++j2){
                        for (int k2 = k + c[i][j][k]; k2 <
                           tiempo; ++k2) {
                            ex2 += x[h][i2][j2][k2];
                        }
                    }
                }
                modelo.addGenConstrIndicator(x[h][i][j][k],
```

```
                    1, tiempo * ex1 >= ex2);
            }
        }
    }
}

// El dron no puede salir de un lugar distinto al que llegó
   después de recorrer el arco
for(int h = 0; h < num_drones; ++h){
    for(int i = 0; i < n + 1; ++i){
        for(int j = 0; j < n + 1; ++j){
            for(int k = 0; k < tiempo; ++k){
                GRBLinExpr ex;
                for(int i2 = 0; i2 < n + 1; ++i2){
                    if(i2 != j){
                       for(int j2 = 0; j2 < n + 1; ++j2){
                            if (k + c[i][j][k] < tiempo) {
                                ex += x[h][i2][j2][k + c[i][j
                                   ][k]];
                            }
                       }
                    }
                }
                modelo.addGenConstrIndicator(x[h][i][j][k],
                   1, ex == 0);
            }
        }
    }
}

// Si el dron recorre un arco que no lo lleva a la base,
   entonces la energía que queda es la que tenía menos la que
    usó para recorrer el arco.
for(int h = 0; h < num_drones; ++h){
    for(int i = 0; i < n + 1; ++i){
```

```
            for(int j = 1; j < n + 1; ++j){
                for(int k = 0; k < tiempo; ++k){
                    if (k + c[i][j][k] < tiempo) {
                        modelo.addGenConstrIndicator(x[h][i][j][k
                           ], 1, e[h][k + c[i][j][k]] == e[h][k]
                           - c[i][j][k]);
                    }
                }
            }
        }
    }

    // Si el dron recorre un arco que lo lleva a la base,
       entonces la energía se recarga
    for(int h = 0; h < num_drones; ++h){
        for(int i = 0; i < n + 1; ++i){
            for(int k = 0; k < tiempo; ++k){
                if (k + c[i][0][k] < tiempo) {
                   modelo.addGenConstrIndicator(x[h][i][0][k], 1,
                       e[h][k + c[i][0][k]] == b);
                }
            }
        }
    }

    // El dron comienza con toda la batería
    for(int h = 0; h < num_drones; ++h){
        modelo.addConstr(e[h][0] == b);
    }

    // El dron debe tener suficiente energía para recorrer el
       arco en el instante en el que lo hará
    for(int h = 0; h < num_drones; ++h){
        for(int k = 0; k < tiempo; ++k){
            GRBLinExpr ex;
```

```
            for(int i = 0; i < n + 1; ++i){
                for(int j = 0; j < n + 1; ++j){
                    ex += x[h][i][j][k] * c[i][j][k];
                }
            }
            modelo.addConstr(e[h][k] - ex >= 0);
        }
    }

    // Se calculará la suma de todos los tiempos de viaje
       incurridos por el dron
    for(int h = 0; h < num_drones; ++h){
        GRBLinExpr ex;
        for(int i = 0; i < n + 1; ++i){
            for(int j = 0; j < n + 1; ++j){
                for(int k = 0; k < tiempo; ++k){
                    ex += x[h][i][j][k] * c[i][j][k];
                }
            }
        }
        modelo.addConstr(t[h] == ex);
    }

    GRBVar varmax = modelo.addVar(0, +GRB_INFINITY, 0,
       GRB_INTEGER, formato("varmax"));
    std::vector<GRBVar> temp;
    for(int i = 0; i < num_drones; ++i){
        temp.push_back(t[i]);
    }
    modelo.addGenConstrMax(varmax, &temp[0], num_drones);

    GRBLinExpr objetivo;
    objetivo += varmax;
    modelo.setObjective(objetivo, GRB_MINIMIZE);
```

```
modelo.write("modelo.lp");
modelo.optimize( );
if (modelo.get(GRB_IntAttr_Status) == GRB_OPTIMAL){
    modelo.write("modelo.sol");
    std::vector<std::vector<std::array<int, 3>>> rutas;
    for (int h = 0; h < num_drones; ++h) {
       std::vector<std::array<int, 3>> ruta;
       for (int k = 0; k < tiempo; ++k) {
          for (int i = 0; i < n + 1; ++i) {
             for (int j = 0; j < n + 1; ++j) {
                if (x[h][i][j][k].get(GRB_DoubleAttr_X) >
                   0.5) {
                   ruta.push_back({ k, i, j });
                }
             }
          }
       }
       if (!ruta.empty( )) {
          rutas.push_back(ruta);
       }
    }
    std::cout << rutas.size( ) << "\n";
    for (auto& ruta : rutas) {
       std::cout << ruta.size( ) << "\n";
       for (auto [k, i, j] : ruta) {
          std::cout << k << "␣" << i << "␣" << j << "\n";
       }
    }
} else if (modelo.get(GRB_IntAttr_Status) == GRB_UNBOUNDED){
    std::cout << "Modelo␣no␣acotado\n";
} else if (modelo.get(GRB_IntAttr_Status) == GRB_INFEASIBLE){
    std::cout << "Modelo␣infactible\n";
    modelo.computeIIS( );
    modelo.write("modelo.ilp");
} else {
```

```
        std::cout << "Status no manejado\n";
    }
} catch (const GRBException& e){
    std::cout << e.getMessage( ) << "\n";
}

// g++ -std=c++23 -O3 modelo.cpp -lgurobi_c++ -lgurobi100 -o
   modelo
// ./modelo
```

A.2. Generador de instancias.

```
#include <algorithm>
#include <array>
#include <chrono>
#include <ctime>
#include <iostream>
#include <random>
#include <vector>

int semilla( ){
   return std::chrono::duration_cast<std::chrono::microseconds>(
      std::chrono::system_clock::now( ).time_since_epoch( )).
      count( );
}

double distancia(double x1, double x2, double y1, double y2) {
    return sqrt(pow(x2 - x1 , 2) + pow(y2 - y1, 2));
}

int representante(int repr[], int i) {
   if (repr[i] != i) {
      repr[i] = representante(repr, repr[i]);
   }
   return repr[i];
}
```

```
bool unidos(int repr[], int i, int j) {
   return representante(repr, i) == representante(repr, j);
}

void une(int repr[], int i, int j) {
   repr[representante(repr, i)] = j;
}

int main( ) {
    int n;
    std::cin >> n;

    std::mt19937_64 gen(semilla( ));
    std::uniform_real_distribution <double> dist(0.0, n);
    std::uniform_int_distribution<int> entero(1, ceil(n/3));

    double puntos[n+1][2];
    for (int i = 0; i < n+1; ++i) {
        puntos[i][0] = dist(gen);
        puntos[i][1] = dist(gen);
    }

    std::vector<std::array<int, 2>> v;
    for (int i = 1; i < n + 1; ++i) {
       for (int j = i + 1; j < n + 1; ++j) {
          v.push_back({ i, j });
       }
    }
    std::shuffle(v.begin( ), v.end( ), gen);
    int repr[n + 1], aristas[n - 1][2], pos = 0;
    std::iota(repr, repr + n + 1, 0);
    for (int i = 0; i < v.size( ); ++i) {
        if (!unidos(repr, v[i][0], v[i][1])) {
            une(repr, v[i][0], v[i][1]);
```

```
            aristas[pos][0] = v[i][0];
            aristas[pos][1] = v[i][1];
            ++pos;
        }
    }

    int distancias[n + 1][n + 1];
    for(int i = 0; i < n + 1; ++i){
        for(int j = i; j < n + 1; ++j){
            distancias[i][j] = lround(distancia(puntos[i][0],
               puntos[j][0], puntos[i][1], puntos[j][1]));
            distancias[j][i] = distancias[i][j];
        }
    }

    int T = 0, D = 0;
    for(int i = 0; i < n - 1; ++i){
        int suma = distancias[0][aristas[i][0]] + distancias[
           aristas[i][0]][aristas[i][1]] + distancias[aristas[i
           ][1]][0];
        T += suma;
        D = std::max(D, suma);
    }

    int B = lround((pow(n, 0.25)) * 2 * D);
    std::cout<< n << "␣" << B << "␣" << 2*T << "\n";
    for (int i = 0; i < n + 1; ++i) {
        std::cout << puntos[i][0] << "␣" << puntos[i][1] << "\n";
    }
    std::uniform_real_distribution <double> factor(0.5, 2);
    for(int k = 0; k < 2*T; ++k){
        for(int i = 0; i < n+1; ++i){
            for(int j = 0; j < n + 1; ++j){
                std::cout<<std::max(1,(int)lround(distancias[i][j
                   ]*factor(gen)))<<"␣";          // garantizar
```

```
                que no hay costos 0 (el dron siempre gasta
                energía porque permanece volando, aunque no
                cambie de vértice)
            }
            std::cout<<"\n";
        }
    }
    for(int i = 0; i < n - 1; ++i){
        std::cout<<aristas[i][0]<<" "<<aristas[i][1]<<"\n";
    }
}
```

A.3. Heurística 1 - Mejor solución factible transformada.

```
#include <algorithm>
#include <iostream>
#include <math.h>
#include <vector>

struct punto{
   double x, y;
};

struct arista{
   int i, j;
};

int main(){
   // Leer la entrada:
   int n, b, tiempo;       // número de vértices (sin contar la
      base), batería, tiempo de simulación
   std::cin>> n >> b >> tiempo;

   punto arr[n + 1];       // coordenadas de los puntos (no se
      usarán, pero vendrán en la entrada)
   for (int i = 0; i < n + 1; ++i){
```

```
    std::cin >> arr[i].x >> arr[i].y;
}

int c[n+1][n+1][tiempo];     // Matriz de costos de cada
    tiempo
for(int k = 0; k < tiempo; ++k){
    for(int i = 0; i < n + 1; ++i){
        for(int j = 0; j < n + 1; ++j){
            std::cin >> c[i][j][k];
            c[i][j][k] = std::max(1, c[i][j][k]);
        }
    }
}

int aristas[n-1][2];    // Aristas
for(int i = 0; i < n - 1; ++i){
    std::cin>> aristas[i][0] >> aristas[i][1];
}

int num_drones = lround(sqrt(n));    // por ahora, el número
    de drones no formará parte de la entrada, sino que se
    decidirá de forma independienteg

std::vector<punto> ciclo[num_drones]; // Guardar el camino
    cerrado (ciclo) de los drones.

std::vector<arista> sol[num_drones];
int h_sin_viento;
std::cin >> h_sin_viento;
for(int k = 0; k < num_drones; ++k){
    int ciclo, id_dron;
    std::cin >> id_dron >> ciclo;
    arista aux;
    std::cin >> aux.j;
    for(int i = 1; i < ciclo; ++i){
```

```
            aux.i = aux.j;
            std::cin >> aux.j;
            sol[k].push_back(aux);
        }
    }

    // Evaluar la solución
    std::vector <int> h(num_drones);
    for(int k = 0; k < num_drones; ++k){
        int b_actual = b;
        int tiempos[sol[k].size()];  // Aquí se guardara en que
            tiempo se evalua ir de i a j
        int t = 0;
        for(int i = 0; i < sol[k].size(); ++i){
            arista v = sol[k][i];
            tiempos[i] = t;
            if(b_actual - c[v.i][v.j][t] <= 0){
                while(b_actual - c[v.i][0][t] < 0){ // Mientras no
                    sea posible regresar a la base con la bateria
                    actual:
                    --i;                            // Se regresa un
                        tiempo atras
                    v = sol[k][i];
                    t = tiempos[i];
                    b_actual += c[v.i][v.j][t];
                }
                t += c[v.i][0][t];
                t += c[0][v.i][t];
                b_actual = b - c[0][v.i][t];
                tiempos[i] = t;
            }
            b_actual -= c[v.i][v.j][t];
            t += c[v.i][v.j][t];
        }
        h[k] = t;
```

```
    }

    std::vector<int>::iterator r = std::max_element(h.begin(), h.
        end());
    std::cout << *r << "\n";

    for(int i = 0; i < num_drones; ++i){
        std::cout<< i + 1 << "\n" << sol[i].size() + 1 << "\n" << 0
            << "␣";
        for(auto p: sol[i]){
            std::cout<< p.j << "␣";
        }
        std::cout<<"\n";
    }

    //std::cout << h[0] << " " << h[1];
}
```

A.4. Heurística 2 - Algoritmo glotón sin comunicación entre drones.

```
#include <algorithm>
#include <chrono>
#include <iostream>
#include <math.h>
#include <queue>
#include <set>
#include <vector>

struct evento {
    int tiempo, dron, bateria, vertice;
};

struct punto{
    double x, y;
```

```
};

struct dis{
   int vertice, distancia;
};

struct arista{
   int i, j;
};

struct ver_dis{
   int vertice, distancia;
};

double distancia(double x1, double x2, double y1, double y2) {
   return sqrt(pow(x2 - x1 , 2) + pow(y2 - y1, 2));
}

bool operator<(evento a, evento b) {                    // el
   evento a importa menos que el b?
   return a.tiempo > b.tiempo;     // el evento a importa menos
      si ocurre despues
}

int main(){
   // Lectura de la entrada

   int n, b, tiempo;       // número de vértices (sin contar la
      base), batería, tiempo de simulación
   std::cin>> n >> b >> tiempo;

   punto arr[n + 1];       // coordenadas de los puntos (no se
      usarán, pero vendrán en la entrada)
   for (int i = 0; i < n + 1; ++i){
      std::cin >> arr[i].x >> arr[i].y;
```

```
}

int c[n+1][n+1][tiempo];     // Matriz de costos de cada
   tiempo
for(int k = 0; k < tiempo; ++k){
   for(int i = 0; i < n + 1; ++i){
      for(int j = 0; j < n + 1; ++j){
         std::cin >> c[i][j][k];
         c[i][j][k] = std::max(1, c[i][j][k]);
      }
   }
}

std::set<int> requeridas[n+1];
arista aristas[n-1];    // Aristas
for(int i = 0; i < n - 1; ++i){
   std::cin>> aristas[i].i >> aristas[i].j;
   requeridas[aristas[i].i].insert(aristas[i].j);
   requeridas[aristas[i].j].insert(aristas[i].i);

}

int num_drones = lround(sqrt(n));    // por ahora, el número
   de drones no formará parte de la entrada, sino que se
   decidirá de forma independienteg

std::vector<punto> ciclo[num_drones]; // Guardar el camino
   cerrado (ciclo) de los drones.

auto t0 = std::chrono::high_resolution_clock::now( );

// Heurística 2:
std::vector <int> c_p(n + 1);    // Matriz de costos para
   calcular p
for(int j = 0; j < n + 1; ++j){
```

```
    c_p[j] = lround(distancia(arr[0].x, arr[j].x, arr[0].y, arr
       [j].y));
}
std::vector<int>::iterator r = std::max_element(c_p.begin(),
   c_p.end());
int p = *r *2;

std::vector <dis> d; // Saber cual vertice está más cerca
for(int i = 0; i < n-1; ++i){   // Se calcula la distancia de
   0->i y de 0->j y se guarda la menor en d
    dis aux1, aux2;
    aux1.vertice = aristas[i].i;
    aux1.distancia = c[0][aristas[i].i][0];
    aux2.vertice = aristas[i].j;
    aux2.distancia = c[0][aristas[i].j][0];
    d.push_back(aux1.distancia < aux2.distancia ? aux1 : aux2);
}
std::sort(d.begin(), d.end(),[](dis a, dis b){
    return a.distancia < b.distancia;
});

std::priority_queue<evento> cp;
int inicio = d[0].distancia;
for(int i = 0; i < num_drones; ++i){   // LLenar la cola de
   prioridad con los eventos
    int tiempo, bateria, vertice;
    tiempo = d[i].distancia;    // La distancia al vertice más
       cercano
    bateria = b - d[i].distancia;
    vertice = d[i].vertice;
    cp.push(evento{tiempo, i, bateria, vertice});
    punto aux;
    aux.x = 0;
    aux.y = vertice;
    ciclo[i].push_back(aux);
```

```
}

int h_max = 0;
while(!cp.empty()){ // Evaluar todos los eventos.
   evento actual = cp.top();
   cp.pop();
   if(actual.bateria <= p){   // Si la batería es menor a P,
      el dron debe regresar a la base.
      punto aux;
      aux.x = actual.vertice;
      aux.y = 0;
      ciclo[actual.dron].push_back(aux);
      actual.tiempo += c[actual.vertice][0][actual.tiempo];
      actual.bateria = b;
      actual.vertice = 0;
      cp.push(actual);
   }else{   // El dron tiene más de P de batería
      if (requeridas[actual.vertice].size( ) > 0){ // ¿el vé
         rtice v tiene aristas pendientes?
         std::vector <ver_dis> destinos;
         for(auto e: requeridas[actual.vertice]){  // Ciclo
            para llenar los vertices destinos (incidentes de
            una arista requerida) y saber cual es el más
            cercano
            ver_dis v;
            v.vertice = e;
            v.distancia = c[actual.vertice][e][actual.tiempo];
            destinos.push_back(v);
         }
         std::sort(destinos.begin(), destinos.end(),[](ver_dis
             a, ver_dis b){
            return a.distancia < b.distancia;
         });
         punto aux;
         aux.x = actual.vertice;
```

```
            aux.y = destinos[0].vertice;
            ciclo[actual.dron].push_back(aux);
            requeridas[actual.vertice].erase(destinos[0].vertice)
               ;
            requeridas[destinos[0].vertice].erase(actual.vertice)
               ;
            actual.bateria -= c[actual.vertice][destinos[0].
               vertice][actual.tiempo];
            actual.tiempo += c[actual.vertice][destinos[0].
               vertice][actual.tiempo];
            actual.vertice = destinos[0].vertice;
            cp.push(actual);
         }else{   // el vértice v no tiene aristas pendientes
            std::vector <ver_dis> destinos;
            for(int j = 1; j < n+1; ++j){ // Ciclo para obtener
               todos los vertices de las aristas pendientes
               for(auto e: requeridas[j]){
                  ver_dis v;
                  v.vertice = e;
                  v.distancia = c[actual.vertice][e][actual.
                     tiempo];
                  destinos.push_back(v);
               }
            }
            if(destinos.empty()){   // Si ya no hay aristas
               pendientes (por tanto tampoco vertices) se detiene
                el ciclo sin antes devolver al evento actual a la
                cola de prioridad.
               punto aux;
               aux.x = actual.vertice;
               aux.y = 0;
               ciclo[actual.dron].push_back(aux);
               actual.tiempo += c[actual.vertice][0][actual.
                  tiempo];
               actual.bateria = b;
```

```
            actual.vertice = 0;
         }else{   // Si aún hay aristas pendientes se
            selecciona el vértice más cercano
            std::sort(destinos.begin(), destinos.end(),[](
               ver_dis a, ver_dis b){
               return a.distancia < b.distancia;
            });
            // No se eliminan o mejor dicho no existen porque
               si el vertice x no tiene aristas requeridas,
               entonces no hay un numero "y" en él. y al reves
               .
            //requeridas[actual.vertice].erase(destinos[0].
               vertice);
            //requeridas[destinos[0].vertice].erase(actual.
               vertice);
            punto aux;
            aux.x = actual.vertice;
            aux.y = destinos[0].vertice;
            ciclo[actual.dron].push_back(aux);
            actual.bateria -= c[actual.vertice][destinos[0].
               vertice][actual.tiempo];
            actual.tiempo += c[actual.vertice][destinos[0].
               vertice][actual.tiempo];
            actual.vertice = destinos[0].vertice;
            cp.push(actual);
         }
      }
   }
   if(actual.tiempo > h_max){
      h_max = actual.tiempo;
   }
}
std::cout<< h_max << "\n";
for(int i = 0; i < num_drones; ++i){
   std::cout<< i + 1 << "\n" << ciclo[i].size() + 1 << "\n" <<
```

```
            0 << "␣";
        for(auto p: ciclo[i]){
            std::cout<< p.y << "␣";
        }
        std::cout<< "\n";
    }

    auto t1 = std::chrono::high_resolution_clock::now( );
    std::cout<< std::chrono::duration<double>(t1 - t0).count( ) <<
        "␣segundos\n";
}
```

A.5. Heurística 3 - Algoritmo glotón con comunicación entre drones.

```
#include <algorithm>
#include <chrono>
#include <iostream>
#include <math.h>
#include <queue>
#include <set>
#include <vector>

struct evento {
    int tiempo, dron, bateria, vertice;
};

struct punto{
    double x, y;
};

struct dis{
    int vertice, distancia;
};
```

```
struct arista{
   int i, j;
};

struct ver_dis{
   int vertice, distancia;
};

double distancia(double x1, double x2, double y1, double y2) {
   return sqrt(pow(x2 - x1 , 2) + pow(y2 - y1, 2));
}

bool operator<(evento a, evento b) {                    // el
   evento a importa menos que el b?
   return a.tiempo > b.tiempo;     // el evento a importa menos
      si ocurre despues
}

int main(){
   // Lectura de la entrada

   int n, b, tiempo;       // número de vértices (sin contar la
      base), batería, tiempo de simulación
   std::cin>> n >> b >> tiempo;

   punto arr[n + 1];       // coordenadas de los puntos (no se
      usarán, pero vendrán en la entrada)
   for (int i = 0; i < n + 1; ++i){
      std::cin >> arr[i].x >> arr[i].y;
   }

   int c[n+1][n+1][tiempo];     // Matriz de costos de cada
      tiempo
   for(int k = 0; k < tiempo; ++k){
      for(int i = 0; i < n + 1; ++i){
```

```
        for(int j = 0; j < n + 1; ++j){
            std::cin >> c[i][j][k];
            c[i][j][k] = std::max(1, c[i][j][k]);
        }
    }
}

std::set<int> requeridas[n+1];
arista aristas[n-1];    // Aristas
for(int i = 0; i < n - 1; ++i){
    std::cin>> aristas[i].i >> aristas[i].j;
    requeridas[aristas[i].i].insert(aristas[i].j);
    requeridas[aristas[i].j].insert(aristas[i].i);

}

int num_drones = lround(sqrt(n));    // por ahora, el número
    de drones no formará parte de la entrada, sino que se
    decidirá de forma independienteg

std::vector<punto> ciclo[num_drones]; // Guardar el camino
    cerrado (ciclo) de los drones.

auto t0 = std::chrono::high_resolution_clock::now( );

// Heurística 2:
std::vector <int> c_p(n + 1);    // Matriz de costos para
    calcular p
for(int j = 0; j < n + 1; ++j){
    c_p[j] = lround(distancia(arr[0].x, arr[j].x, arr[0].y, arr
        [j].y));
}
std::vector<int>::iterator r = std::max_element(c_p.begin(),
    c_p.end());
int p = *r *2;
```

```
std::vector <dis> d; // Saber cual vertice está más cerca
for(int i = 0; i < n-1; ++i){   // Se calcula la distancia de
   0->i y de 0->j y se guarda la menor en d
   dis aux1, aux2;
   aux1.vertice = aristas[i].i;
   aux1.distancia = c[0][aristas[i].i][0];
   aux2.vertice = aristas[i].j;
   aux2.distancia = c[0][aristas[i].j][0];
   d.push_back(aux1.distancia < aux2.distancia ? aux1 : aux2);
}
std::sort(d.begin(), d.end(),[](dis a, dis b){
   return a.distancia < b.distancia;
});

std::priority_queue<evento> cp;
int inicio = d[0].distancia;
for(int i = 0; i < num_drones; ++i){   // LLenar la cola de
   prioridad con los eventos
   int tiempo, bateria, vertice;
   tiempo = d[i].distancia;    // La distancia al vertice más
      cercano
   bateria = b - d[i].distancia;
   vertice = d[i].vertice;
   cp.push(evento{tiempo, i, bateria, vertice});
   punto aux;
   aux.x = 0;
   aux.y = vertice;
   ciclo[i].push_back(aux);
}

int h_max = 0;
while(!cp.empty()){ // Evaluar todos los eventos.
   evento actual = cp.top();
   cp.pop();
```

```
if(actual.bateria <= p){   // Si la batería es menor a P,
   el dron debe regresar a la base.
   punto aux;
   aux.x = actual.vertice;
   aux.y = 0;
   ciclo[actual.dron].push_back(aux);
   actual.tiempo += c[actual.vertice][0][actual.tiempo];
   actual.bateria = b;
   actual.vertice = 0;
   cp.push(actual);
}else{   // El dron tiene más de P de batería
   if (requeridas[actual.vertice].size( ) > 0){ // ¿el vé
      rtice v tiene aristas pendientes?
      std::vector <ver_dis> destinos;
      for(auto e: requeridas[actual.vertice]){  // Ciclo
         para llenar los vertices destinos (incidentes de
         una arista requerida) y saber cual es el más
         cercano
         ver_dis v;
         v.vertice = e;
         v.distancia = c[actual.vertice][e][actual.tiempo];
         destinos.push_back(v);
      }
      std::sort(destinos.begin(), destinos.end(),[](ver_dis
          a, ver_dis b){
         return a.distancia < b.distancia;
      });
      punto aux;
      aux.x = actual.vertice;
      aux.y = destinos[0].vertice;
      ciclo[actual.dron].push_back(aux);
      requeridas[actual.vertice].erase(destinos[0].vertice)
         ;
      requeridas[destinos[0].vertice].erase(actual.vertice)
         ;
```

```
            actual.bateria -= c[actual.vertice][destinos[0].
               vertice][actual.tiempo];
            actual.tiempo += c[actual.vertice][destinos[0].
               vertice][actual.tiempo];
            actual.vertice = destinos[0].vertice;
            cp.push(actual);
         }else{   // el vértice v no tiene aristas pendientes
            std::vector <ver_dis> destinos;
            for(int j = 1; j < n+1; ++j){ // Ciclo para obtener
               todos los vertices de las aristas pendientes
               for(auto e: requeridas[j]){
                  ver_dis v;
                  v.vertice = e;
                  v.distancia = c[actual.vertice][e][actual.
                     tiempo];
                  destinos.push_back(v);
               }
            }
            if(destinos.empty()){   // Si ya no hay aristas
               pendientes (por tanto tampoco vertices) se detiene
                el ciclo sin antes devolver al evento actual a la
                cola de prioridad.
               punto aux;
               aux.x = actual.vertice;
               aux.y = 0;
               ciclo[actual.dron].push_back(aux);
               actual.tiempo += c[actual.vertice][0][actual.
                  tiempo];
               actual.bateria = b;
               actual.vertice = 0;
            }else{   // Si aún hay aristas pendientes se
               selecciona el vértice más cercano
               std::sort(destinos.begin(), destinos.end(),[](
                  ver_dis a, ver_dis b){
                  return a.distancia < b.distancia;
```

```
            });
            // No se eliminan o mejor dicho no existen porque
               si el vertice x no tiene aristas requeridas,
               entonces no hay un numero "y" en él. y al reves
               .
            //requeridas[actual.vertice].erase(destinos[0].
               vertice);
            //requeridas[destinos[0].vertice].erase(actual.
               vertice);
            punto aux;
            aux.x = actual.vertice;
            aux.y = destinos[0].vertice;
            ciclo[actual.dron].push_back(aux);
            actual.bateria -= c[actual.vertice][destinos[0].
               vertice][actual.tiempo];
            actual.tiempo += c[actual.vertice][destinos[0].
               vertice][actual.tiempo];
            actual.vertice = destinos[0].vertice;
            cp.push(actual);
         }
      }
   }
   if(actual.tiempo > h_max){
      h_max = actual.tiempo;
   }
}
std::cout<< h_max << "\n";
for(int i = 0; i < num_drones; ++i){
   std::cout<< i + 1 << "\n" << ciclo[i].size() + 1 << "\n" <<
       0 << "␣";
   for(auto p: ciclo[i]){
      std::cout<< p.y << "␣";
   }
   std::cout<< "\n";
}
```

```
    auto t1 = std::chrono::high_resolution_clock::now( );
    std::cout<< std::chrono::duration<double>(t1 - t0).count( ) <<
        "␣segundos\n";
}
```

A.6. Sistema gráfico.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width,␣initial-
            scale=1.0">
        <title>Jovanni</title>
        <script type="text/javascript" src="dom.js" async></script>
        <style>
            svg {
                width: 750px;
                height: 750px;
                border: 1px solid black;
            }

            svg * {
                transition-property: all;
                transition-timing-function: linear;
            }

            .vertice {
                fill: red;
                r: 14px;
                cursor: move;
            }

            .arista {
                stroke: blue;
```

```
    stroke-width: 2px;
}

.arista_visitada {
    stroke: #D8D8FF;
}

.texto {
    font-size: 20px;
}

.texto_dron {
    fill: white;
    z-index: 20;
}

.dron {
    width: 16px;
    height: 16px;
}
```

```
.contenedor {
  display: flex;
  flex-direction: row;
  flex-wrap: wrap;
}

#contenedor > div {
  width: 50%;
  text-align:center;
}

.botones{
        margin: 20px auto;
        display: flex;
```

```
                        justify-content: center;
                }

                input[type=number] {
                        width: 100px;
                        margin: 0px 0px 0px 10px;
                }

                textarea {
                        resize: none;
                }

                button {
                        margin: 0px 10px 0px 10px;
                }

    </style>
</head>
<body onload="main( )">
            <div>
            Introduce el número de vértices: <input type="
               number" id="vertices" min="5" value="5"> <
               button id="generar" onclick=genera_instancia()
               >Generar</button>
            </div>
            <div class="contenedor" style='margin: 20px;'>
                    <div>
                            <svg id="svg" viewBox="0 0 1000
                               1000"></svg>
                            <div class="botones">
                                    <button id="mover_drones"
                                        hidden="hidden">Mueve
                                        drones generados</
                                       button>
                                    <label id="div_velocidad"
```

```
                        hidden="hidden"><
                        input id="velocidad"
                        type="range" min="0.2"
                         max="5" value="1"
                        step="0.2" onchange=
                        cambiar_velocidad()>
                        Velocidad</label>
        </div>
</div>
<div style='margin: 20px 0px 0px 40px'>
        <textarea id="leer_instancia"
           cols="50" rows="20"></textarea
           >
        <div class="botones">
                <button id="borrar"
                   onclick=
                   borrar_instancia()>
                   Borrar instancia</
                   button>
                <button onclick=
                   lee_instancia()>Cargar
                    instancia</button>
        </div>

        <textarea id="instancia" cols="50
           " rows="20" readonly="readonly
           "></textarea>
        <div class="botones">
                <button id="copiar"
                   onclick=
                   copiar_instancia()
                   hidden="hidden">Copiar
                    instancia</button>
        </div>
</div>
```

```
            </div>
            <div class="contenedor">
                    <div id="modifica_puntos"></div>
                    <div id="modifica_aristas"></div>
                    <div id="caminos"></div>
            </div>

<script>
                    let velocidad = 1;
                    function cambiar_velocidad(){
                            velocidad = document.
                               getElementById("velocidad").
                               value;
                    }

   // utilidades generales

                    function distancia(x1, x2, y1, y2){
                            return Math.sqrt(Math.pow(x2 - x1
                                , 2) + Math.pow(y2 - y1, 2));
                    }

                    function generador_aleatorio(semilla) {
                            return function( ) {
                                    return semilla = semilla
                                       * 48271 % 2147483647;
                            }
                    }

                    function distribucion_lineal(gen, min,
                       max) {
       return function( ) {
          return gen( ) * (max - min) / (2147483647 - 1) +
             min;
       }
```

```
            }

            function matriz(...dimensiones){
   function crea_matriz_impl(indice, dimensiones) {
      if (indice == dimensiones.length) {
         return undefined;
      } else {
         let arr = new Array(dimensiones[indice]);
         for (let i = 0; i < arr.length; ++i) {
            arr[i] = crea_matriz_impl(indice + 1,
               dimensiones);
         }
         return arr;
      }
   }
   return crea_matriz_impl(0, dimensiones);
            }

class union_pertenencia {
   constructor(n) {
      this.repr = Array.from({ length: n }, (v, i) =>
         i);
   }
   representante(i) {
      if (this.repr[i] != i) {
         this.repr[i] = this.representante(this.repr[i])
            ;
      }
      return this.repr[i];
   }
   unidos(i, j) {
      return this.representante(i) == this.representante
         (j);
   }
   une(i, j) {
```

```
            this.repr[this.representante(i)] = j;
        }
    }

    function posicion_mouse(evento) {
        let ctm = document.getElementById("svg").getScreenCTM
            ( );
        return { "x": (evento.clientX - ctm.e) / ctm.a, "y":
            (evento.clientY - ctm.f) / ctm.d };
    }

    // utilidades de svg

    let id_vertice = 0, id_dron = 0;

    function reinicia_dibujo( ) {
        id_vertice = 0, id_dron = 0;
        for (let tipo of [ "circle", "line", "rect", "text"
            ]) {
            for (let elem of Array.from(document.
                getElementsByTagName(tipo))) {
                elem.remove( );
            }
        }
    }

    function dibuja_vertice(x, y) {
        let elem = create_svg("circle", { "id": `v${
            id_vertice}`, "configure_properties": [ { "
            aristas1": [ ], "aristas2": [ ] } ], "
            configure_attributes": [ { "class": "vertice", "cx
            ": x, "cy": y } ] });
        elem.texto = create_svg("text", { "id": `t${
            id_vertice}`, "configure_properties": [ { "
            textContent": id_vertice } ], "
```

```
      configure_attributes": [ { "class": "texto", "x":
      x + 20, "y": y + 5 } ] });
   document.getElementById("svg").appendChild(elem);
   document.getElementById("svg").appendChild(elem.texto
      );
   ++id_vertice;
   return elem;
}

function dibuja_arista(v1, v2) {
   let elem = create_svg("line", { "id": `${Math.min(v1.
      id.substring(1), v2.id.substring(1))}--${Math.max(
      v1.id.substring(1), v2.id.substring(1))}`, "
      configure_attributes": [ { "class": "arista", "x1"
      : v1.getAttribute("cx"), "y1": v1.getAttribute("cy
      "), "x2": v2.getAttribute("cx"), "y2": v2.
      getAttribute("cy") } ] });
   v1.aristas1.push(elem), v2.aristas2.push(elem);
   document.getElementById("svg").appendChild(elem);
   return elem;
}

function dibuja_dron(vertice) {
   let elem = create_svg("rect", { "id": `d${id_dron}`,
      "configure_attributes": [ { "class": "dron" } ] })
      ;
   elem.texto = create_svg("text", { "id": `T${id_dron
      }`, "configure_properties": [ { "textContent":
      id_dron + 1 } ], "configure_attributes": [ { "
      class": "texto texto_dron" } ] });
   document.getElementById("svg").appendChild(elem);
   document.getElementById("svg").appendChild(elem.texto
      );
   mueve_dron(elem, null, vertice, 0);
   ++id_dron;
```

```
    return elem;
}

function mueve_dron(dron, origen, destino, duracion) {
    duracion = Math.max(duracion, 0.001);
    dron.style.transitionDuration = `${duracion}s`;
    dron.setAttributeNS(null, "x", Number(destino.
        getAttribute("cx")) - 8);
    dron.setAttributeNS(null, "y", Number(destino.
        getAttribute("cy")) - 8);
    if (origen != null) {
        let extremo1 = Number(origen.id.substring(1)),
            extremo2 = Number(destino.id.substring(1));
        let arista = document.getElementById(`${Math.min(
            extremo1, extremo2)}--${Math.max(extremo2,
            extremo1)}`);
        if (arista != null) {
            arista.style.transitionDuration = `${duracion}s
                `;
            arista.classList.add("arista_visitada");
        }
    }
    let animacion1 = create_svg("animate", { "
        configure_attributes": [ { "attributeName": "x", "
        to": Number(destino.getAttribute("cx")) - 6, "dur"
        : `${duracion}s`, "fill": "freeze" } ] });
    let animacion2 = create_svg("animate", { "
        configure_attributes": [ { "attributeName": "y", "
        to": Number(destino.getAttribute("cy")) + 6, "dur"
        : `${duracion}s`, "fill": "freeze" } ] });
    dron.texto.appendChild(animacion1);
    dron.texto.appendChild(animacion2);
    animacion1.beginElement( ), animacion2.beginElement(
        );
}
```

```
                function mueve_dron_secuencia(dron,
                   vertices, duraciones) {
    for (let arista of document.querySelectorAll(".
       arista_visitada")) {
       arista.style.transitionDuration = "0s";
       arista.classList.remove("arista_visitada");
    }
    mueve_dron(dron, null, vertices[0], 0);
                        for (let i = 0, tiempo = 0; i <
                           vertices.length - 1; tiempo +=
                            1000 * (duraciones[i++] /
                           velocidad)) {
                                window.setTimeout(
                                   mueve_dron, tiempo,
                                   dron, vertices[i],
                                   vertices[(i + 1) %
                                   vertices.length], (
                                   duraciones[i] /
                                   velocidad));
                        }
                }

function indices_a_vertices(indices) {
    return indices.map((indice) => document.
       getElementById(`v${indice}`));
}

// generación y lectura de instancia

let n, B, T, semilla, instancia_cargada = false, puntos
    = [], ct = [], aristas = [], distancias = [],
    recorrido = [], duracion_dron = [];

                function genera_grafica_base(){
```

```
                        n = parseInt(document.
                           getElementById("vertices").
                           value) + 1, semilla = 2 * Math
                           .random();
                        puntos = matriz(0), aristas =
                           matriz(0), distancias = matriz
                           (n, n);

                        // Generar los puntos aleatorios
                        for(let i = 0; i < n; ++i){
                                puntos.push([Math.random
                                   () * n, Math.random()
                                   * n]);
                        }

                        // Revolver las aristas de la grá
                           fica completa
                        let posibles = [];
                        for(let i = 1; i < n; ++i){
                                for(let j = i + 1; j < n;
                                    ++j){
                                        posibles.push([i,
                                           j]);
                                }
                        }
                        posibles = posibles.sort((a, b) =
                           > 0.5 - Math.random());
                                // std::shuffle

// Elegir las aristas del árbol
let repr = new union_pertenencia(n);
                        for(let arista of posibles) {
                                if (!repr.unidos(arista
                                   [0], arista[1])){
                                        repr.une(arista
```

```
                                    [0], arista
                                    [1]);
                                aristas.push(
                                    arista);
                        }
                    }

                    // Calcular la matriz de
                       distancias
                    for(let i = 0; i < n; ++i){
                            for(let j = i; j < n; ++j
                               ){
                                    distancias[i][j]
                                       = Math.round(
                                       distancia(
                                       puntos[i][0],
                                       puntos[j][0],
                                       puntos[i][1],
                                       puntos[j][1]))
                                       ;
distancias[j][i] = distancias[i][j];
                            }
                    }
            }

            function completa_grafica_windy(){
                    T = 0; let D = 0;
                    for(let i = 0; i < n - 2; ++i){
                            let temp = distancias[0][
                               aristas[i][0]] +
                               distancias[aristas[i
                               ][0]][aristas[i][1]] +
                                distancias[aristas[i
                               ][1]][0];
                            T += temp, D = Math.max(D
```

```
                                        , temp);
                                }

                                B = Math.round((Math.pow(n, 0.25)
                                    ) * 2 * D), ct = matriz(n, n,
                                    2 * T);
                                let gen_rango =
                                    distribucion_lineal(
                                    generador_aleatorio(semilla),
                                    0.5, 2);
                                for(let k = 0; k < 2 * T; ++k){
                                        for(let i = 0; i < n; ++i
                                            ){
                                                for(let j = 0; j
                                                    < n; ++j){
                                                        ct[i][j][
                                                            k] =
                                                            Math.
                                                            max(1,
                                                             Math.
                                                            round(
                                                            distancias
                                                            [i][j]
                                                             *
                                                            gen_rango
                                                            ( )));
                                                }
                                        }
                                }
                        }

        function lee_grafica_windy(texto){
            // revisar: hay muchas cosas que no se validan

                                let arr = texto.trim( ).split(/\s
```

```
   +/).map(s => parseFloat(s));

// Revisar la no negatividad de
   todos los valores
for(let actual of arr){
        if(actual < 0){
                return false;
        }
}
n = parseInt(arr[0]) + 1;
B = parseInt(arr[1]);
T = parseInt(arr[2]) / 2;
tam_correcto = 3 + (2 * n) + (2 *
    T * n * n) + ((n - 2) * 2);
if(tam_correcto != arr.length - 1
    && tam_correcto != arr.length
   ){
        return false;
}

// Obtener los puntos de la
   instancia
puntos.length = n;
let pos = 3;
for(let i = 0; i < n; ++i){
        puntos[i] = new Array(2);
        puntos[i][0] = arr[pos
           ++];
        puntos[i][1] = arr[pos
           ++];
}

// Obtener los valores de ct de
   la instancia
ct = matriz(n, n, 2 * T);
```

```
                        for(let k = 0; k < 2 * T; ++k){
                                for(let i = 0; i < n; ++i
                                   ){
                                        for(let j = 0; j
                                           < n; ++j){
                                                ct[i][j][
                                                   k] =
                                                   arr[
                                                   pos
                                                   ++];
                                        }
                                }
                        }

                        // Obtener las aristas requeridas
                            de la instancia
                        aristas.length = n - 2;
                        for(let i = 0; i < n - 2; ++i){
                                aristas[i] = new Array(2)
                                   ;
                                aristas[i][0] = arr[pos
                                   ++];
                                aristas[i][1] = arr[pos
                                   ++];
                        }

    return true;
}

// rutinas auxiliares de interfaz gráfica

                // Sirve para mostrar la interfaz donde
                   se puede modificar los valores de un v
                   értice
                function modifica_puntos(num){
```

```
                        let div = create_html("div", {"id
                           ": `p${num}`, "innerHTML": `Vé
                           rtice ${num}`});
                        let x = create_html("input", {"id
                           ": `p${num}x`, "type":"number"
                           , "min": 0, "max": n});
                        let y = create_html("input", {"id
                           ": `p${num}y`, "type":"number"
                           , "min": 0, "max": n});
                        let boton = create_html("button",
                            {"id": `b${num}`, "innerHTML"
                           : "Modificar", "onclick":
                           function(){
                                /*if(x.value < 0 || x.
                                   value > n || y.value <
                                    0 || y.value > n){
                                        return window.
                                           alert("Ingrese
                                           un valor
                                           entre 0 y " +
                                           n);
                                }*/
                                if(instancia_cargada && !
                                   confirm("Usted cargó
                                   una instancia; al
                                   modificar la posición
                                   de un vértice la
                                   instancia cambiara
                                   totalmente por el
                                   sistema. ¿Desea
                                   continuar con los
                                   cambios?")){
    return;
                                }
instancia_cargada = false;
```

```
                                    puntos[num] = [ x.value,
                                       y.value ];
                                    for(let i = 0; i < n; ++i
                                       ){
                                          distancias[i][num
                                             ] = Math.round
                                             (distancia(
                                             puntos[i][0],
                                             puntos[num
                                             ][0], puntos[i
                                             ][1], puntos[
                                             num][1]));
                                          distancias[num][i
                                             ] = distancias
                                             [i][num];
                                    }
                                    completa_grafica_windy(),
                                        actualiza_instancia()
                                       ;
                              }});

        x.value = puntos[num][0], y.value = puntos[num][1];
                              div.appendChild(x), div.
                                 appendChild(y), div.
                                 appendChild(boton);
                              document.getElementById("
                                 modifica_puntos").appendChild(
                                 div);
                        }

                        // Función para mostrar la interfaz donde
                            se pueden modificar las aristas
                           requeridas
                        function modifica_aristas(num){
                              let div = create_html("div", {"id
```

```
    ": `e${num}`, "innerHTML": `
    Arista ${num + 1}`});
let ei = create_html("input", {"
    id": `e${num}i`, "type":"
    number", "min": 1, "max": n
    -1});
let ej = create_html("input", {"
    id": `e${num}j`, "type":"
    number", "min": 1, "max": n
    -1});
let boton = create_html("button",
     {"innerHTML": "Modificar", "
    onclick": function(){
          if(ei.value < 1 || ei.
             value > n-1 || ej.
             value < 1 || ej.value
             > n-1 || ei.value ==
             ej.value){
                return window.
                   alert("Ingrese
                   una arista
                   valida entre 1
                   y " + (n-1) +
                    " además que
                   ambas aristas
                   deben ser
                   distintas.");
          }
          aristas[num] = [ ei.value
             , ej.value ];
          completa_grafica_windy(),
              actualiza_instancia()
             ;
}});
```

```
        ei.value = aristas[num][0], ej.value = aristas[num
           ][1];
                        div.appendChild(ei), div.
                           appendChild(ej), div.
                           appendChild(boton);
                        document.getElementById("
                           modifica_aristas").appendChild
                           (div);
                    }

                    function ingresa_recorrido(textarea, num)
                       {
                            let arr = textarea.value.split(/\
                               s+/).map(s => parseInt(s));
                            // Validar que esten entre el
                               rango [0,n]
                            let rango_valido = true;
                            for(let i of arr){
                                    if(i < 0 || i > n - 1){
                                            rango_valido =
                                               false;
                                            break;
                                    }
                            }

                            // Validar que comienza y termina
                                en cero
                            if(arr[0] != 0 || arr[arr.length
                               - 1] != 0 || !rango_valido){
                                    window.alert(`Ingrese un
                                       recorrido válido para
                                       el dron ${num}. Rango
                                       [0,n] donde comienza y
                                        termina en 0.`);
        return false;
```

```
}

// Instanciar el recorrido
recorrido[num] = matriz(arr.
   length);
for(let i = 0; i < arr.length; ++
   i){
        recorrido[num][i] = arr[i
           ];
}

// Extraer las costos del
   recorrido conforme a ct
let tiempo = 0;
duracion_dron[num] = matriz(arr.
   length);
for(let i = 0; i <= recorrido[num
   ].length - 2; ++i){
        duracion_dron[num][i] =
           ct[recorrido[num][i]][
           recorrido[num][i +
           1]][tiempo];
        tiempo += ct[recorrido[
           num][i]][recorrido[num
           ][i + 1]][tiempo];
        //console.log("i = " +
           recorrido[num][i] + "
           j = " + recorrido[num
           ][i + 1] + " ct = " +
           ct[recorrido[num][i]][
           recorrido[num][i +
           1]][tiempo] +" tiempo
           = " + tiempo);
}
duracion_dron[num][recorrido[num
```

```
                ].length - 1] = 0;
        return true;
            }

            function modifica_camino(num){
                /// Esta parte es donde se
                   ingresará la solución, aun
                   esta en desarrollo
                let div = create_html("div");
                div.innerHTML = "Ingrese el
                   recorrido del dron " + (num +
                   1);
                let textarea = create_html("input
                   ", {"id": `walk_d${num}`, "
                   type": "text", "style": "
                   margin: 0px 0px 0px 10px;"});
                let boton = create_html("button",
                    {"innerHTML": "Ingresar", "
                   onclick": function(){
                        ingresa_recorrido(
                           textarea, num);
                }});
                div.appendChild(textarea);
                div.appendChild(boton);
                document.getElementById("caminos"
                   ).appendChild(div);
            }

            function actualiza_instancia_textarea(){
                let temp = `${n-1} ${B} ${2 * T}\
                   n`;
                for(let i = 0; i < n; ++i){
                        temp += `${puntos[i][0]}
                           ${puntos[i][1]}\n`;
                }
```

```
                        for(let k = 0; k < 2 * T; ++k){
                                for(let i = 0; i < n; ++i
                                   ){
                                        for(let j = 0; j
                                           < n; ++j){
                                                temp += `
                                                   ${ct[i
                                                   ][j][k
                                                   ]} `;
                                        }
                                        temp += `\n`;
                                }
                        }
                        for(let i = 0; i < aristas.length
                           ; ++i){
                                temp += `${aristas[i][0]}
                                    ${aristas[i][1]}\n`;
                        }
                        document.getElementById("
                           instancia").value = temp;
                }

        function actualiza_instancia_controles(){
                        document.getElementById("
                           modifica_puntos").innerHTML =
                           "";
                        for(let i = 0; i < puntos.length;
                            ++i){
                                modifica_puntos(i);
                        }
                        document.getElementById("
                           modifica_aristas").innerHTML =
                            "";
                        for(let i = 0; i < aristas.length
                           ; ++i){
```

```
                modifica_aristas(i);
            }
            document.getElementById("caminos"
               ).innerHTML = "";
            let num_drones = Math.round(Math.
               sqrt(n));
            ////////////////////////////////////////

            recorrido = matriz(num_drones);
            duracion_dron = matriz(num_drones
               );
            ////////////////////////////////////////

            for(let i = 0; i < num_drones; ++
               i){
                modifica_camino(i);
            }
}

function actualiza_instancia_svg(){
            reinicia_dibujo();
            let escala = 1000 / n;
               // Esta escala depende del
               tamaño del textarea

            let vertices = new Array(n);
            for(let i = 0; i < n; ++i){
                vertices[i] =
                   dibuja_vertice(puntos[
                   i][0] * escala, 1000 -
                    puntos[i][1] * escala
                   );
            }

            let e = new Array(n - 2);
```

```
                        for(let i = 0; i < n - 2; ++i){
                                e[i] = dibuja_arista(
                                   vertices[aristas[i
                                   ][0]], vertices[
                                   aristas[i][1]]);
                        }

                        let num_drones = Math.round(Math.
                           sqrt(n));
                        let d = new Array(num_drones);
                        for(let i = 0; i < num_drones; ++
                           i){
                                d[i] = dibuja_dron(
                                   vertices[0]);
                        }
                        document.getElementById("
                           mover_drones").onclick =
                           function( ){
                                for(let i = 0; i <
                                   num_drones; ++i){
                                        if (!
                                           ingresa_recorrido
                                           (document.
                                           getElementById
                                           (`walk_d${i}`)
                                           , i)) {
                    return;
                }
            }
            for(let i = 0; i < num_drones; ++i){
                mueve_dron_secuencia(d[i], indices_a_vertices(
                    recorrido[i]), duracion_dron[i]);
                                }
        };
    }
```

```
function actualiza_instancia(){
   actualiza_instancia_textarea();
   actualiza_instancia_controles();
   actualiza_instancia_svg();
}

                function muestra_botones(){
                        document.getElementById("
                           mover_drones").hidden = false;
                        document.getElementById("
                           div_velocidad").hidden = false
                           ;
                        document.getElementById("copiar")
                           .hidden = false;
                }

// botones principales de la interfaz gráfica

function genera_instancia(){
                        instancia_cargada = false;
                        genera_grafica_base(),
                           completa_grafica_windy();
                        actualiza_instancia();
   muestra_botones();
}

                function lee_instancia(){
                        instancia_cargada = true;
                        if (!lee_grafica_windy(document.
                           getElementById("leer_instancia
                           ").value)) {
      return alert("Ingrese una instancia válida");
   }
                        actualiza_instancia();
```

```
                        muestra_botones();
                }

                function copiar_instancia(){
    navigator.clipboard.writeText(document.getElementById
       ("instancia").value);
                }

                function borrar_instancia(){
                        document.getElementById("
                           leer_instancia").value = "";
                }

// función inicial de prueba

function main( ) {
                        /*let v = [
      dibuja_vertice(500, 500, 0),
      dibuja_vertice(200, 300, 1),
      dibuja_vertice(50, 800, 2),
      dibuja_vertice(500, 660, 3),
   ];
                        let e = [
      dibuja_arista(v[1], v[2]),
      dibuja_arista(v[1], v[3]),
      dibuja_arista(v[2], v[3])
   ];

   let dron = dibuja_dron(v[0]);
                        document.getElementById("p_camino
                           ").onclick = function(){
                                mueve_dron_secuencia(dron
                                   , indices_a_vertices([
                                    0, 1, 2, 3, 0 ]), [
                                   1, 0.5, 2, 0.3, 1 ]);
```

```
                                    };*/

// drag&drop de svg

const svg = document.getElementById("svg");
const min_x = svg.viewBox.baseVal.x;
const max_x = svg.viewBox.baseVal.width + min_x;
const min_y = svg.viewBox.baseVal.y;
const max_y = svg.viewBox.baseVal.height + min_y;

let seleccionado = null;
let offset, transformacion, min_dx, max_dx, min_dy,
   max_dy;

svg.onmousedown = function(evento) {
   if (evento.target.tagName == "circle") {
      if(instancia_cargada && !confirm("Usted cargó
         una instancia; al modificar la posición de
         un vértice la instancia cambiara totalmente
         por el sistema. ¿Desea continuar con los
         cambios?")){
         return;
      }
      instancia_cargada = false;
      seleccionado = evento.target;
   }
};
svg.onmousemove = function(evento) {
   if (seleccionado != null) {
      evento.preventDefault( );
      let aux = posicion_mouse(evento), num =
         seleccionado.id.substring(1), escala = 1000
         / n;
      document.getElementById(`p${num}x`).value = aux
         .x / escala;
```

```
                document.getElementById(`p${num}y`).value =
                    (1000 - aux.y) / escala;
                seleccionado.setAttribute("cx", aux.x),
                    seleccionado.setAttribute("cy", aux.y);
                seleccionado.texto.setAttribute("x", aux.x +
                    20), seleccionado.texto.setAttribute("y",
                    aux.y + 5);
                for (let i of [ 1, 2 ]) {
                    for (let arista of seleccionado[`aristas${i
                        }`]) {
                        arista.setAttribute(`x${i}`, aux.x);
                        arista.setAttribute(`y${i}`, aux.y);
                    }
                }
            }
        };
        svg.onmouseup = svg.onmouseleave = function(evento) {
            if (seleccionado != null) {
                let aux = posicion_mouse(evento), num =
                    seleccionado.id.substring(1), escala = 1000
                    / n;
                document.getElementById(`p${num}x`).value = aux
                    .x / escala;
                document.getElementById(`p${num}y`).value =
                    (1000 - aux.y) / escala;
                document.getElementById(`b${num}`).onclick();
                seleccionado = null;
            }
        };
    }
  </script>
 </body>
</html>
```

Bibliografía

[1] Benavent, E., A. Carrotta, A. Corberán, J. M. Sanchis y D. Vigo: *Lower bounds and heuristics for the Windy Rural Postman Problem.* European Journal of Operational Research, 176(2):855–869, 2007.

[2] Benavent, E., A. Corberán, I. Plana y J. M. Sanchis: *Min-Max K-vehicles Windy Rural Postman Problem.* Networks, 54:216 – 226, Dic. 2009.

[3] Benavent, E., A. Corberán y J. M. Sanchis: *A metaheuristic for the min–max windy rural postman problem with K vehicles.* Computational Management Science, 7:269–287, 2010.

[4] Campbell, J. F., Ángel Corberán, I. Plana y J. M. Sanchis: *Drone arc routing problems.* Networks, 72(4):543–559, 2018.

[5] Corberán, A., I. Plana y J. M. S. Sanchis: *A branch & cut algorithm for the windy general routing problem and special cases.* Networks, 49(4):245–257, 2007.

[6] Corberán, A., I. Plana y J. M. S. Sanchis: *The Windy General Routing Polyhedron: A Global View of Many Known Arc Routing Polyhedra.* SIAM Journal on Discrete Mathematics, 22(2):606–628, 2008.

[7] Eiselt, H. A., M. Gendreau y G. Laporte: *Arc Routing Problems, Part II: The Rural Postman Problem.* Operations Research, 43(3):399–414, 1995.

[8] Frederickson, G. N., M. S. Hecht y C. E. Kim: *Approximation algorithms for some routing problems.* En *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, págs. 216–227, 1976.

[9] Gurobi Optimization, LLC: *Gurobi Optimizer Reference Manual*, 2020. `http://www.gurobi.com`.

[10] Lenstra, J. K. y A. H. G. R. Kan: *On general routing problems.* Networks, 6(3):273–280, 1976.

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

[12] Moshe Dror, Helman Stern, P. T.: *Postman tour on a graph with precedence relation on arcs.* Networks, 17:283–294, 1987.

[13] Nossack, J., B. Golden, E. Pesch y R. Zhang: *The windy rural postman problem with a time-dependent zigzag option.* European Journal of Operational Research, 258(3):1131–1142, 2017.

[14] Pearn, W. L.: *Solvable cases of the k-person Chinese postman problem.* Operations Research Letters, 16(4):241–244, 1994.

[15] Wu, B. Y., H. L. Wang, S. Ta Kuan y K. M. Chao: *On the uniform edge-partition of a tree.* Discrete Applied Mathematics, 155(10):1213–1223, 2007.