Datos del estudiante

Nombre y apellidos Toni Ballesteros Martínez

Fecha de entrega 03/01/2026

Actividad 2. Planificación multiobjetivo de rutas de drones en entornos urbanos

Introducción

Para la elaboración de la práctica he usado JavaScript como lenguaje de programación.
La estructura de carpetas e información de ejecución puede verse en el *readme* del repositorio publicado en https://github.com/bftoni/algoritmos_avanzados

Modelado

Tenemos un mapa de la ciudad representado como un conjunto de puntos conectados entre sí. Cada punto tiene coordenadas* en el plano y puede ser de tres tipos:

- Hub: almacén central donde inicia y termina la ruta
- Delivery: o puntos de entrega que el dron debe visitar
- Charger: estaciones de recarga de batería

* En nuestro caso las coordenadas iniciales (0,0) parten de la izquierda inferior izquierda

El dron puede volar directamente entre cualquier par de puntos, pero cada trayecto tiene dos costes asociados:

- Distancia: longitud en línea recta entre los puntos
- Riesgo: peligrosidad del vuelo (vamos a suponer que es el terreno el que puede condicionar esta variable)

Además, en el mapa existen como mínimo 3 zonas prohibidas de vuelo (no-fly zones) representadas como polígonos que el dron no puede atravesar.

El objetivo es encontrar una ruta que visite todos los puntos exactamente una vez, regresando al origen, y que simultáneamente minimice tanto la distancia total como el riesgo acumulado.

Restricciones

Las únicas restricciones que debemos tener en cuenta para los vuelos son:

- Evitar zonas prohibidas: Ningún segmento de vuelo puede cruzar o tocar una zona no-fly. Si un trayecto directo entre dos puntos atraviesa una zona prohibida, ese trayecto no es válido.
- Limitación de batería*: El dron tiene capacidad limitada de batería. Debe poder completar el tramo entre dos puntos de recarga consecutivos sin quedarse sin energía

* En nuestro caso vamos a suponer que una unidad de distancia es una unidad de batería

Cálculo de Costes y optimización

Para cada trayecto entre dos puntos calculamos:

- La distancia como la línea recta que los une (teorema de Pitágoras aplicado a sus coordenadas)
- El riesgo, en nuestro caso hemos puesto una función lineal, riesgo = 0.05 × distancia + 0.5, este último valor es para aportar un riesgo mínimo a las distancias más cortas. Las distancias más largas tendrán un riesgo proporcionalmente mayor.

De las soluciones encontradas, buscamos un conjunto de soluciones válidas llamado frontera de Pareto.

Una ruta A es mejor que una ruta B si:

- A tiene menor o igual distancia Y menor o igual riesgo que B

- Y al menos en uno de los dos aspectos A es estrictamente mejor

Descripción de algoritmos

Antes de comentar los diferentes algoritmos, comento las funciones que son comunes a lo largo del código y son las funciones del tema 7 pero en JS:

Métodos comunes usados por los 3 algoritmos

distance(a, b)

Calcula la distancia euclidiana entre a y b aplicando el teorema de Pitágoras a sus coordenadas.

weight(a, b)

Retorna los pesos vectoriales de una arista: distancia, riesgo (0.05×distancia+0.5) y coste de batería.

edgeFeasible(a, b)

Verifica si una arista es factible (no cruza zonas no-fly). Usa caché para evitar recálculos.

completeWeights

Genera la matriz completa de pesos para todas las aristas factibles del grafo.

routeFeasible(path, graph)

Valida si el path es factible verificando geometría y capacidad de batería en cada segmento.

evaluate(path, graph)

Calcula los objetivos de una ruta: distancia total, riesgo acumulado y número de recargas necesarias.

dominates(a, b)

Determina si la solución 'a' domina a 'b' en el sentido de Pareto (mejor o igual en todos los objetivos, estrictamente mejor en al menos uno).

paretoFront(solutions)

Filtra un conjunto de soluciones para retornar solo las no dominadas (frontera de Pareto).

Detallo a continuación cada una de las implementaciones algorítmicas.

Backtracking / Branch-and-Bound

Algoritmo de búsqueda exhaustiva que explora sistemáticamente todas las rutas posibles mediante backtracking. Funciona construyendo la ruta nodo por nodo, descartando caminos que violan restricciones (zonas no-fly o batería insuficiente).

Tambien descarta tempranamente caminos cuando una arista cruza zonas prohibidas, la batería es insuficiente, o no quedan nodos factibles alcanzables.

Algoritmo geométrico (geo_heuristic)

Heurística constructiva con verificación geométrica de factibilidad y mejora local 2-opt. Genera rutas rápidamente seleccionando iterativamente el nodo más cercano factible. Usamos también una solución “random” para generar posibles caminos.
En nuestro código, nombramos los métodos principales:

nearestFeasibleRoute(graph, start, randomize)

Construye una ruta desde el nodo inicial seleccionando en cada paso el siguiente nodo. Si randomize=false, elige siempre el más cercano (greedy puro). Si randomize=true, selecciona aleatoriamente entre los 3 candidatos más cercanos factibles.

twoOpt(route, graph, maxIter)

Aplica mejora local intercambiando pares de aristas para reducir distancia. Verifica que las nuevas aristas no violen restricciones geométricas. Se ejecuta hasta maxIter iteraciones o hasta que no haya mejoras.

solve(graph, opts)

Función principal que genera múltiples soluciones: una ruta greedy pura y varias rutas randomizadas (por defecto 5 en total). Aplica 2-opt a cada una y retorna la frontera de Pareto del conjunto completo.

Metaheurística

Implementación de NSGA-II que evoluciona una población de rutas mediante operadores genéticos. Explora el espacio de soluciones combinando recombinación, mutación y selección basada en dominancia de Pareto.

En nuestro código, nombramos los métodos principales:

greedyFeasible(graph, start)

Genera una solución inicial mediante construcción greedy, seleccionando siempre el nodo más cercano factible. Sirve como semilla de calidad para la población inicial.

orderCrossover(p1, p2)

Operador de cruce que combina dos rutas padres preservando subsecuencias válidas. Copia un segmento del primer padre y completa con genes del segundo en orden, evitando duplicados.

swapMutation(p)

Operador de mutación que intercambia aleatoriamente dos nodos en la ruta, introduciendo variación genética en la población.

tournament(pop)

Selección por torneo que compara dos individuos aleatorios y retorna el dominante según Pareto. Si ninguno domina, elige aleatoriamente.

makeRoute(perm, start)

Construye un circuito hamiltoniano válido insertando el nodo inicial al principio y final de una permutación de nodos intermedios.

solve(graph, opts)

Al igual que para los algoritmos anteriores, esta es la función principal que inicializa población (combinando semilla greedy y rutas aleatorias), ejecuta generaciones evolutivas aplicando cruce y mutación, y mantiene solo individuos factibles. Retorna la frontera de Pareto final tras las iteraciones configuradas.

Setup Experimental

Para probar los experimentos he creado cuatro instancias en formato JSON (dentro de la carpeta data/instances) con la siguiente estructura:

- **Capacidad de batería (battery_capacity):** Indica la capacidad de batería disponible (una unidad de distancia = 1 de batería en nuestro caso)
- **Ciudades o puntos de entrega/recarga (nodes):** array de nodos indicando posición (x,y), tipo “hub o delivery” y un booleano indicando si se puede recargar ahí o no (is_charger)
- **Zonas de no-vuelo (no-fly):** array de coordenadas [x,y] que definen los vértices del polígono sobre los cuales no se puede sobrevolar.

He puesto un ejemplo de un mapa generado para 15 instancias en la raíz del proyecto (plot_n15.svg)

Las instancias tienen 10, 15, 20, y 25 puntos respectivamente y el nivel de batería inicialmente se ha establecido en N=10, 60 para N=15, 70 para N=20, y 80 para N=25.

Las métricas evaluadas en cada experimentos son el tiempo de ejecución en ms, el hipervolumen la diversidad y el tamaño frontera o soluciones óptimas.

Resultados

Algoritmo	N	Tiempo (ms)	Hipervolumen	Diversidad	Tamaño Frontera
exact_bb	10	66.58	1553.46	~0	2
exact_bb	15	30000.06	962.31	0	1
exact_bb	20	30000.09	191.93	0	1
exact_bb	25	30000.19	**0**	0	**0**
geo_heuristic	10	**0.4**	1553.46	~0	2
geo_heuristic	15	**0.4**	**1021.88**	0	1
geo_heuristic	20	**0.4**	**610.99**	0	1
geo_heuristic	25	**1.4**	**261.10**	0	1
metaheuristic	10	3838.0	1553.46	0	1
metaheuristic	15	2527.4	1021.88	0	1
metaheuristic	20	1596.8	390.36	0	1
metaheuristic	25	1430.6	87.18	0	1

* Los resultados se pueden consultar en la carpeta results, dentro de data.

* * También las gráficas generadas, dentro de report/figs

Discusión

exact_bb muestra el comportamiento esperado de un algoritmo exhaustivo: escala exponencialmente alcanzando el timeout (30s) para N≥15. Solo es viable en N=10

(~67ms), pero garantiza optimalidad. Para N=25 agota el tiempo sin explorar suficiente espacio, resultando en frontera vacía.

geo_heuristic es el que tiene el rendimiento más óptimo con tiempos prácticamente constantes (~0.4-1.4ms) independientemente del tamaño. Supera por 1000× al metaheuristic y 75000× al exact_bb en N=10, manteniéndose el más rápido en todas las instancias.

metaheuristic el tiempo decrece con N (3838ms en N=10 → 1430ms en N=25). Esto sugiere convergencia prematura o dificultad para generar población inicial diversa en instancias grandes, requiriendo ajuste del parámetro maxTries para mantener exploración efectiva.

Conclusiones

El algoritmo exacto (Branch & Bound) garantiza soluciones óptimas pero fracasa por timeout en instancias N≥15 debido a crecimiento exponencial.

La heurística geométrica, sin embargo, demostró ser la opción superior, combinando tiempos de ejecución extremadamente bajos (<2ms) con la mejor calidad de soluciones en instancias medianas y grandes

Metaheurística requiere calibración: NSGA-II mostró convergencia prematura en instancias grandes, tal vez jugando más con la población y/o generaciones podríamos haber sacado mejores resultados.