Research Article

INTERNATIONAL JOURNAL OF ADVANCED ROBOTIC SYSTEMS

International Journal of Advanced Robotic Systems
January-February 2021: 1–16

DOI: 10.1177/17298814219895551
journals.sagepub.com/home/arx
SAGE

Flight planning in multi-unmanned aerial vehicle systems: Nonconvex polygon area decomposition and trajectory assignment

Georgy Skorobogatov, Cristina Barrado, Esther Salamí and Enric Pastor

Abstract

Nowadays, it is quite common to have one unmanned aerial vehicle (UAV) working on a task but having a team of UAVs is still rare. One of the problems that prevent us from using teams of UAVs more frequently is flight planning. In this work, we present the first open-source solution (https://pypi.org/project/pode/) for splitting any complex area into multiple parts. The area of interest can be convex or nonconvex and can include any number of no-flight zones. Four solutions, based on the algorithm of Hert and Lumelsky, are tested with the aim of improving the compactness of the partitions. We also show how the shape of the partitions influences flight performance in a real case scenario.

Keywords

Unmanned aerial vehicle, multi-UAV, unmanned aerial system, coverage path planning, trajectory planning

Date received: 8 June 2020; accepted: 4 January 2021

Topic Area: Mobile Robots and Multi-Robot Systems
Topic Editor: Nak-Young Chong
Associate Editor: Changjoo Nam

Introduction

Modern unmanned aerial vehicles (UAVs) cost relatively cheap and are less time consuming compared to traditional ways of collecting aerial data, making them a good choice in a wide range of applications. As a drawback, the maximum operating range and flight time are limited by the UAV batteries. One possibility to overcome this limitation is to use a system with several UAVs.[1] A system with multiple UAVs is useful for collecting information in parallel over vast areas, or when there are time constraints, a single unmanned aerial system cannot complete the task. Systems with multiple UAVs are also fault-tolerant and flexible; if one UAV fails, it will not hinder the mission from being completed. Using multiple UAVs also makes it possible to have them in different locations or perform different tasks at the same time.

Since systems of multiple UAVs are still relatively new, there is still work to be done to make such systems more widely used. Currently, to the best of our knowledge, there are no ready solutions to split an area of any complex shape into a variable number of parts. In this work, we present an attempt to fill this gap. We show how to split an area of interest given the number of UAVs, the requirements for the area to be covered by each UAV, and, optionally, the initial position of each UAV. Not all splits are equally valid

Computer Architecture Department, Universitat Politècnica de Calatunya, Barcelona, Spain

Corresponding author:
Georgy Skorobogatov, Computer Architecture Department, Technical University of Catalonia, Castelldefels, 08860, Spain.
Email: georgy.skorobogatov@upc.edu

for their subsequent flight. We use the compactness of the resulting shapes as a maximization target and test four different heuristics. Next, we analyze an example of a mission in which the area has to be covered in a back-and-forth manner with a given cross-track separation. We show the resulting trajectories that are assigned using a ready solution existing in the literature[2] and analyze if the partition can be improved.

This article is organized as follows: The "Related work" section discusses existing works that address similar problems. The "Materials and methods" section describes the proposed flight planning in detail. Next, "Case of study" section shows how to apply flight planning in a real-life scenario. Finally, "Conclusions" section summarizes future work and problems yet to be solved.

Related work

Several research works have been published on the problem of path planning for UAVs. A recent survey[3] on coverage path planning discusses different types of area decomposition and patterns of planned trajectories. This survey also covers a wide range of research works. Some of these works are related specifically to coverage path planning for teams of UAVs. Disregarding the publications discussing path planning with partial information, such as search methods, there are several interesting works that deserve attention.

The work "Multiple UAV cooperative searching operation using polygon area decomposition and efficient coverage algorithms"[4] can be considered as the basis of our work. The authors present an algorithm to divide the area of interest taking into account the initial locations of the UAVs and their relative capabilities. UAVs cover the resulting subareas in a back-and-forth pattern. The algorithm that the authors used for polygon decomposition was taken from the paper "Polygon area decomposition for multiple-robot workspace division."[5] The authors implemented only the part of the algorithm for convex polygons without holes. Unfortunately, this is not always true in real life. In our work, we implement the complete algorithm for any complex polygon with or without holes. We also make the implementation open for public use.

In the work "Aerial remote sensing in agriculture: a practical approach to area coverage and path planning for fleets of mini aerial robots," Barrientos et al.[6] discuss the use of a multiple UAV system that can take georeferenced pictures and creates a full mosaic. One of the main contributions of that work is an automatic one-phase partition manager, which is based on the negotiation between UAVs taking into account their capabilities. When each UAV gets its task, a path planning algorithm determines the best path for each UAV to follow. In this work, the authors used the cellular decomposition of an area of interest and applied a flood-fill algorithm to obtain the subareas. This approach has several drawbacks. For example, this approach will result in partial cells near the edges of the area. The algorithm will also restart when the flood-fill cannot proceed. On the other hand, this algorithm could be considered superior to exact-partition algorithms, specifically for tasks in which UAVs have to visit each cell in their subareas and not intrude on neighboring subareas. The authors did not provide their implementation, hence, it is difficult to properly evaluate this approach.

Another work[7] improves an existing harmony search algorithm, which is a population-based algorithm searching for the best configuration when a stop criterion is met. The algorithm was tested on a vineyard parcel for three quadrotors. This work, similar to the previous one, uses a cellular decomposition, where each cell corresponded to an image sample.

In the work "Area coverage with heterogeneous UAVs using scan patterns," Berger et al.[8] considered the problem of scanning an area with a team of UAVs. The main contribution was the formulation of an optimization problem following the requirement established by an operator. The requirement could be either a minimum flight time or to obtain a high point density within a time limit. In their work, the algorithm ran multiple iterations to optimize for the specified requirement, while in our case, the algorithm calculates the area partition and trajectories in a single run without any optimization.

Other works referenced in the survey[3] also discuss a spiral coverage pattern or a line formation when all UAVs travel in a line. We do not include these works here since we are mainly interested in improving area partitioning.

Materials and methods

Multiflight planner

The purpose of the flight planner is to take the parameters of the fleet of UAVs and the area to cover and assign a subarea to each UAV of the team. There can be additional side goals such as minimizing the duration of the flight or minimizing the number of turns. In this work, we will consider the following input parameters:

i. Area—a polygon defining an area to be covered by a team of UAVs.
ii. No-flight zones—holes of the given polygon.
iii. Number of UAVs—the number of UAVs in the team that will observe the area.
iv. Initial positions of UAVs—departure locations for UAVs trajectories. Initial positions can be either specified by the user or assigned automatically.
v. Cruise speeds—the speed of each UAV in the team. This parameter can affect the resulting partition as faster UAVs will be able to cover larger areas than slower ones.

Algorithm 1. Multiflight planner main algorithm.

```
Data: polygon, IP – initial positions, DF – convex
      divisor function, MP – mission parameters
Result: trajectories
sites = get_sites(IP, MP);
subareas = split(polygon, sites, DF) // Alg. 2;
trajectories = {};
for area in subareas do
    trajectory = build_trajectory(area, MP);
    trajectories.add(trajectory);
end
return trajectories;
```

vi. Field of view of the camera—has a direct effect on the size of the area observed at each moment.
vii. Altitude above ground level—the higher the altitude, the larger the area covered by a sensor. Higher altitude can result in less flight time.
viii. Cross-track overlapping—required if the obtained images have to be stitched into a full map after the flight.

The pseudocode of the main algorithm is presented in Algorithm 1. The algorithm takes as input the following parameters: a polygon defining the area of interest, optional initial positions of UAVs, the remaining input parameters from the list above named *mission parameters*, and a convex divisor function. The purpose of this function is to split the input polygon into convex parts. The mission parameters and the initial positions are then used to build a list of *sites*. A site is a tuple containing the area requirement of a given UAV and, optionally, the corresponding initial positions. How these area requirements are calculated depends on the details of the mission and the characteristics of the UAVs. We will not go into details on how to calculate them. These sites are then used to determine how the input polygon will be split. For each resulting part, the algorithm builds a trajectory that depends on the mission parameters. These trajectories are then returned as the final result of the algorithm.

In the next subsection, how the polygon partition is done considering the area requirements and the initial positions of the UAVs will be explained. After that, in the next subsection, how the trajectories are built within each part of the polygon will be explained.

Polygon partition

The idea of the original algorithm[5] can be briefly summarized as follows. First, the algorithm divides the given polygon into a set of convex parts. These parts are then used to create a so-called region-adjacency graph. The parts are then processed in a specific order. Depending on several factors, different techniques are used to split each part, either for further processing or to return it as a resulting part. Unfortunately, insufficient information is provided regarding many details. Here, we present a more elaborate explanation with several changes that, in our opinion, remove the ambiguity and clarify some critical points. Algorithm 2 presents the pseudocode of the polygon partition function and Figure 1 shows the semantic flowchart, the structured data used, and the links to the other pseudocode parts. The algorithm is explained in the following subsections using Figure 1 as an outline.

Definitions. Before explaining the algorithm of the polygon partition, it is important to clarify some terms. Some of these terms are left the same as in the original paper[5]:

- **Polygon**—a tuple of an exterior contour and a list of interior contours (holes). These contours define an area of interest, where each contour is represented as a list of vertices.
- **Subpolygon**—a perimeter delimiting an area without holes. A subpolygon is represented as a node in the region-adjacency graph.
- **Region-adjacency graph**—graph constructed from the polygon. The nodes of this graph are convex subpolygons. The edges of the graph represent those subpolygons that touch each other by segments. The information about the segments is kept as attributes of the graph edges.
- **Directed graph**—a graph where the edges between the nodes have directions associated with them.
- **Postorder graph traversal**—the way of iterating over nodes of a graph where children are visited first.
- **Predecessors**—the nodes having a path in the directed graph to a specified node.
- **Successors**—the nodes of the directed graph that are reachable from the specified node.
- ***Next neighbor***—the neighbor of a node that is its most immediate successor.
- ***PredPoly***—set containing the specified node and all the nodes that are reachable from it without entering a successor of that node.
- **Site**—a tuple containing the initial position of a UAV in the polygon and the corresponding area requirement.
- **Pseudosite**—site added in the segment between two polygons to connect them. Pseudosites are necessary for the cases when the area requirement of the original site is greater than the area of the polygon where the site is located.

Exact computation. The algorithm for the polygon partition has input parameters the polygon and its sites. Additionally, it receives a convex divisor function that will be explained later. The coordinate values of the polygon and the sites have to be supplied having a "fraction" data type.

Algorithm 2. *split*

Data: polygon, sites, DF - convex divisor function
Result: D – division {sites $\rightarrow$ polygons}
D = {} // Division;
IP = {} // Incomplete parts {sites $\rightarrow$ polygons};
PR = {} // Pseudosites relations {pseudosites $\rightarrow$ sites};
// Alg. 3;
parts = convex_parts(polygon, sites.locations, DF);
graph = to_graph(parts, DF) // Alg. 5;
graph = to_directed(graph);
remaining_nodes = graph.nodes;
while *graph.has_nodes* **do**
  CP = next(remaining_nodes) // Current Polygon;
  $PredPoly$ = graph.pred_poly(CP);
  // Current Sites - Alg. 6;
  CS, sites = select_sites(CP, sites, graph);
  if *len(CS == 0)* **then**
    continue
  R = total_requirements(CS);
  if *len(CS) == 1 and $PredPoly.area < R$* **then**
    // Case 1;
    PS = graph.create_pseudosite(CP, CS);
    graph.remove_nodes_from(*PredPoly*);
    IP.append({CS[0] $\rightarrow$ $PredPoly$});
    PR.append(PS $\rightarrow$ CS[0]);
  else if *len(CS) == 1 and $PredPoly.area == R$* **then**
    graph.remove_nodes_from(*PredPoly*);
    if *$CS[0]$ not in PR* **then**
      // Case 2;
      D.append({CS[0] $\rightarrow$ $PredPoly$});
    else
      // Case 3;
      original_site = PR[CS[0]];
      polygon = union($PredPoly$, IP[original_site]);
      D.append({original_site $\rightarrow$ polygon});
    end
  else
    // Case 4;
    V = graph.reorder_vertices(CP, CS) // Vertices;
    // Alg. 7;
    parts = divide(CP, V, CS, graph);
    target = graph.next_neighbor(CP);
    graph.remove_nodes_from(*PredPoly*);
    // Alg. 9;
    graph = graph.prepend(parts, target);
  end
end
return D;

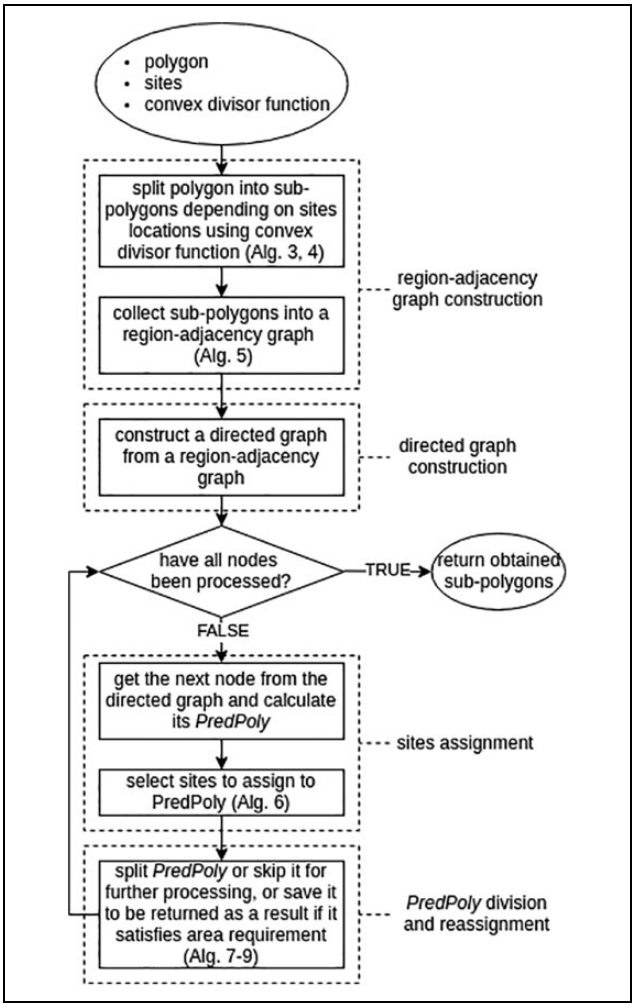

Figure 1. Flowchart of the algorithm that divides a polygon into subpolygons according to the area requirements of each UAV. UAV: unmanned aerial vehicle.

This "fraction" data type is also called "ratio" in some programming languages. The purpose of this type is to keep rational numbers like 1/3 in memory without rounding. Precision errors will appear in cases when there are three or more colinear vertices located on boundaries of polygons and when working with very large and close-to-zero values at the same time. Inexact computation will also lead to accumulating errors in some cases over the course of running the algorithm. Exact geometric computation is a must in our case if we want to have an error-proof robust solution. One has to be careful, though, when using libraries for geometric computations as not all the libraries provide exact computations. For example, the popular library CGAL supports exact computations through a Quotient class but Shapely, GEOS, or JTS do not. CGAL has Python bindings but they do not provide all the necessary functionality. Instead, we used a newly emerged library

Algorithm 3. *convex_parts*

Data: polygon, IP - initial positions, DF - convex divisor function
Result: convex parts of polygon
extra_points = IP - polygon.vertices;
extra_constraints = [polygon.holes, extra_points];
// E.g. Alg. 4;
parts = DF(polygon.border, extra_constraints);
return parts;

gon (https://gon.readthedocs.io, 2019, Azat Ibrakov) implemented purely in Python that enabled us to do the exact computation.

Region-adjacency graph construction. The first step for the polygon partition is to construct the region-adjacency graph from the input (the polygon and the site locations). The paper with the original algorithm[5] did not specify how to do it, so, here, we present our own way. To do that, we have to split the polygon into convex parts, as given in Algorithm 3. Hert and Lumelsky[5] provide a list of references to several different algorithms but do not go into details about which way would be preferable. Such algorithm also must take into account additional constraints. Holes of the polygon must be excluded and the sites' locations have to be placed on the boundaries of the resulting subpolygons. There are very few available implementations and almost all of them do not support exact computations. CGAL contains an implementation of Greene's algorithm[9] but, unfortunately, it is not yet included in Python bindings as of the time of writing this paper. For the purpose of the convex divisor function, we used a constrained Delaunay triangulation[10] from the sect (https://sect.readthedocs.io, 2020, Azat Ibrakov) library. But, as it will be shown later, small convex parts contribute to many sharp angles in the final division. We implemented an extension to the constrained Delaunay triangulation, where consecutive triangles were joined to form larger convex parts. The algorithm is shown in Algorithm 4.

When the separate convex parts are obtained, we need to make a region-adjacency graph out of them. The algorithm for that is shown in Algorithm 5. If the convex partition was performed by Delaunay triangulation, graph construction can be efficiently done in linear time. First, the algorithm creates a mapping of all the segments from the triangles' boundaries to the sets of triangles containing these sides. Then, the graph was created by constructing edges from the sets with two triangles of the aforementioned mapping. If the convex partition was performed by some other algorithm rather than Delaunay triangulation, the algorithm has to iterate over all possible combinations of pairs of convex subpolygons. On each iteration, the intersection of both polygons from the pair is calculated. If the intersection is a segment, the polygon pair is added to the graph as new

Algorithm 4. joined_triangles

Data: polygon, E - extra_points
Result: C – convex parts
extra_constraints = [polygon.holes, E];
triangles = delaunay_triangulation(polygon.border, extra_constraints);
initial_polygon = triangles.pop();
C = [];
while *True* **do**
  resulting_polygon = initial_polygon;
  for *index, polygon* **in** *enumerate(polygons)* **do**
    sides = set(polygon.edges);
    common_side = resulting_polygon.edges ∩ sides;
    if *common_side is None* **then**
      continue
    if *len($E \cup$ common_side.points) > 0* **then**
      continue
    union = resulting_polygon + polygon;
    if *union is Polygon and union.is_convex* **then**
      polygons.pop(index);
      resulting_polygon = union;
  end
  if *resulting_polygon is not initial_polygon* **then**
    initial_polygon = resulting_polygon;
    continue
  C.append(resulting_polygon);
  if *len(polygons) == 0* **then**
    break
  initial_polygon = polygons.pop();
end
return C;

edges. This approach is not optimal time-wise. To make it more efficient, the algorithm for splitting a polygon to convex parts has to return the information on the neighbor of each part. To the best of our knowledge, there are no implementations of such algorithms currently. For graph manipulations, we used NetworkX[11] library. This library allowed us to reorder vertices and obtain successors and ancestors of the nodes. It also allowed us to keep the segments by which the convex subpolygons touch as attributes of graph edges.

Directed graph construction. The second step is to order the nodes of the graph according to the OrderPieces algorithm.[5] The resulting order, in fact, corresponds to a post-order graph traversal.[12] We used a directed graph to keep information about the order. Edges of this graph indicated ancestors and descendants of any given node. The implementation may differ depending on the used library for graph manipulations, hence, we do not include it here. It is important to note that any node can have at most one outcoming edge and any number of incoming edges.

Algorithm 5. *to_graph*

```
Data: parts, DF - convex divisor function
Result: region-adjacency graph
graph = EmptyGraph();
if len(parts) == 1 then
    graph.add_nodes_from(parts);
    return graph;
else
    if DF == delaunay_triangulation then
        parts_per_sides = {} //{side → set(polygon)};
        for part in parts do
            for side in part.sides do
                parts_per_sides[side].add(part);
            end
        end
        for side, parts in parts_per_sides do
            if len(parts) == 2 then
                graph.add_edge(parts, side=side);
        end
    else
        for part, other in combinations_pairs(parts)
          do
            I = part.intersection(other);
            if I is segment then
                graph.add_edge(part, other, I);
        end
    end
end
return graph;
```

Therefore, only those outcoming edges that point to the next neighbors must be left intact and the other edges must be removed. A visual representation of the process of polygon conversion to a region-adjacency graph and from a region-adjacency graph to a directed graph is shown in Figure 2.

Site assignment. Once the directed graph is built, the algorithm iterates over the graph nodes. On each iteration, the algorithm calculates the PredPoly and selects some sites from the initial list of sites. Depending on the selected sites, the algorithm will determine how the PredPoly will be processed. This article presents a contribution to the algorithm for site selection.

This algorithm operates on two lists of sites. The first list is the initial list of sites that were passed as an argument to Algorithm 2 and are still to be assigned. This list either shrinks or stays the same with each iteration over the nodes of the graph. The second list contains the sites that have already been assigned to some of the nodes during previous iterations.

Algorithm 6 shows the algorithm for site selection. From the first list, the algorithm selects the sites that lie inside the subpolygon corresponding to the current node and no any other nodes. It also selects those sites from the second list that were preassigned to the current node. It is possible that, after the execution of the two previous steps, no sites have been selected. In this case, the algorithm selects a single site from the first list. The selected site should be located on the border of the subpolygon corresponding to the current node. If no sites lie in the current node, the algorithm selects a single site from the sites in the first list that do not have a specified location. If such a site exists, the algorithm assigns it a random location inside the subpolygon corresponding to the current node and returns it as a result. We choose to select only one site because selecting more sites will eventually result in more sharp angles in the resulting subpolygons. In the event that the algorithm could not select any site, the current node will be skipped.

PredPoly division and reassignment. Depending on how many sites were selected and the relation between their total area and the area of the PredPoly, four different cases are considered. We will call them cases 1–4.

In case 1, there is only one site and its requirement is greater than the area of *PredPoly*. To satisfy the difference, we create a pseudosite on the boundary between the current subpolygon and its *NextNeighbor*. We keep the information about the relation of the original site with the pseudosite in a mapping of pseudosites to sites. We also keep the information about the current polygon in a mapping of original sites to area-incomplete polygons. The specific location of the pseudosite on the edge was not specified in the description of the algorithm. We choose it to be the center of the edge.

When the area of the *PredPoly* is equal to the area requirement and only one site was assigned to the current node, we check if this site is an original site (case 2) or a pseudosite (case 3). If this is an original site, we save it along with the *PredPoly* so it will be returned as a result later. If the site is a pseudosite, we retrieve the corresponding original site from the mapping of pseudosites to sites. We calculate the geometric union of the *PredPoly* with all the area-incomplete polygons saved previously in the mapping of original sites to area-incomplete polygons. Both the original site and the obtained union are saved to be returned later as a result.

In the case that there is more than one site or the area of the *PredPoly* is greater than the selected site's requirement, the *PredPoly* has to be divided (case 4). Before the division, the current polygon has to be reoriented to a counterclockwise orientation, and all the sites and vertices from the connected neighbors have to be included in the list of vertices. The vertices also have to be ordered in such a way that the edge with the next neighbor would include the first and the last vertices. If there is no next neighbor, then the vertices have to be ordered so that the last vertex would be any site point. This order will allow the neighbor polygons

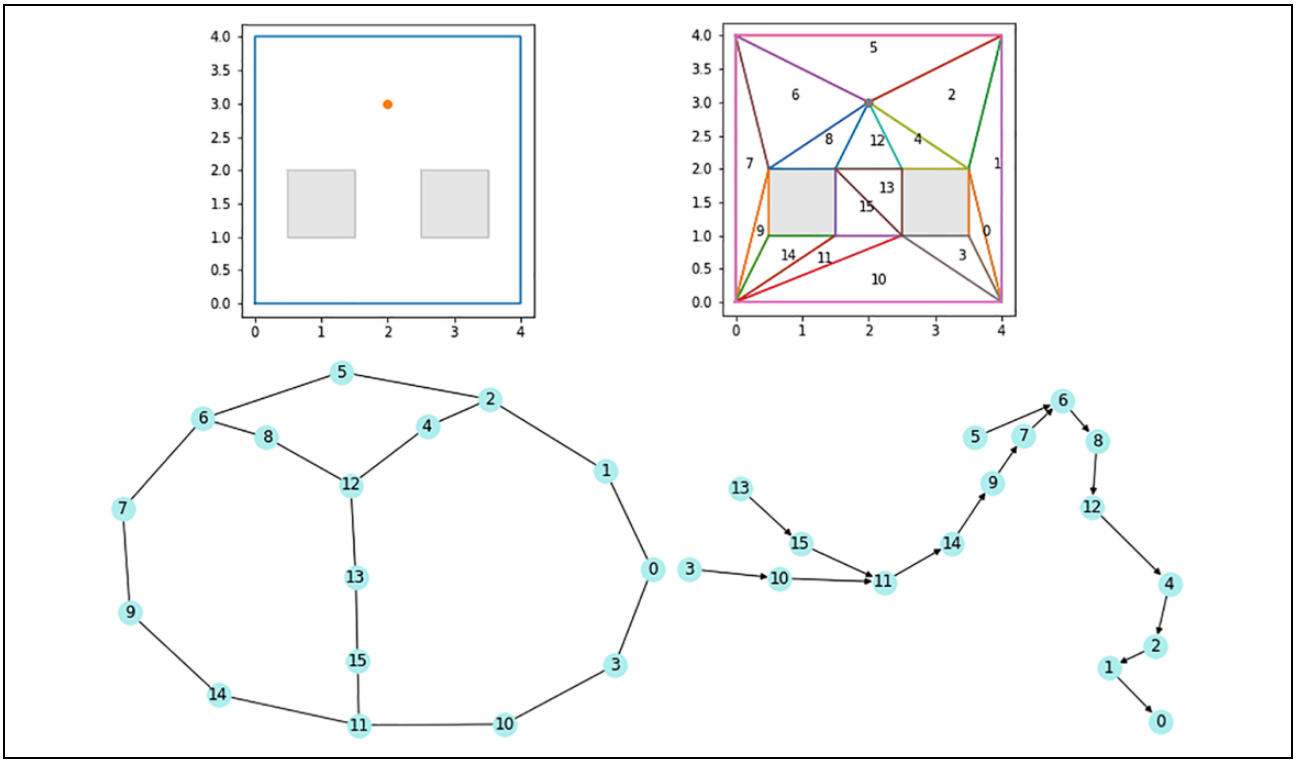

Figure 2. Polygon with two holes and an extra point (upper left image) is split by a constrained Delaunay triangulation to a set of triangles (upper right image). A region-adjacency graph (lower left image) is constructed from this set of triangles. Directed graph corresponding to the postgraph traversal is shown on the lower right image.

to be properly connected when the algorithm will return their union as a result.

The algorithm for division is shown in Algorithm 7. This algorithm iterates over the vertices of the polygon and constructs lines that will split this polygon into left and right parts. For each iteration, the algorithm calculates the sets of *PredPoly* connected to the right parts of the polygon. The algorithm also calculates sites lying on the right parts and their total requirements according to Algorithm 8. The iteration goes on until either the last vertex is encountered or the area of the *PredPoly* on the right part is greater than the total requirement of all the sites on the right part. At the end of the iteration, the algorithm splits the polygon into two parts by a line. The way this line is calculated depends on the relation of the area requirement and the right *PredPoly* corresponding to the last iteration. The endpoint of the line can be either placed in the middle between the last two vertices or calculated by linear interpolation. After the split, the *PredPoly* is split into separate parts, which will be returned as a result. The way the *PredPoly* is split also depends on how its area relates to the area requirement. Different cases are shown in Figure 3.

The algorithm to interpolate the point on the edge of a triangle can be simply derived from the inverse task of a shoelace formula. The *G.sub* function returns a subgraph of the graph with the nodes given as the first argument and the current node given as the second argument. This node is replaced by the polygon given as the third argument. The sites, which are provided as the fourth argument, are assigned to the new node.

After the subgraphs are returned, they have to be prepended back to the graph. An algorithm for prepending is shown in Algorithm 9.

Trajectories assignment

After each UAV gets its own area to cover, trajectories of a back-and-forth pattern can be generated using any appropriate algorithm. In our implementation, we used a ready program.[2] The algorithm behind that program does not perform any time or path length minimization but simply generates the trajectory following input restrictions on the course, track separation, and desired entry and exit sides of the polygon. A screenshot of the program with the map and the GUI panel for the specification of trajectory parameters is shown in Figure 4.

The course direction is chosen along the longest subarea side, as we assume that the resulting number of turns will be less than with any other direction. The track separation is calculated from the width of the observed area. And the width of the observed area can be calculated from equation (1) of "Multiple UAV cooperative searching operation using polygon area decomposition and efficient coverage algorithms"[4]

Algorithm 6. *select_sites*

Data: polygon, sites, graph
Result: sites for polygon division, leftover sites
S_{loc} = {} // set of sites with given locations;
for *site in sites* **do**
  if *site.location is not None* **then**
    S_{loc}.add(site);
end
S_{pre} = {} // sites preassigned in Alg. 6;
for *node in graph* **do**
  for *site in graph[node].sites* **do**
    S_{pre}.add(site);
  end
end
if *len(S_{loc}) == 0 and len(S_{pre}) == 0* **then**
  S_{bare} = [] // sites without locations;
  for *site in sites* **do**
    if *site not in S_{loc}* **then**
      S_{bare}.append(site);
  end
  if *len(S_{bare}) == 0* **then**
    return {}, {};
  location = polygon.border.vertices[0];
  site = Site(sites[-1].area, location);
  return {site}, sites[..-1];
// preassigned sites lying in current polygon;
CS_{pre} = graph[polygon].sites;
ancestors = graph.ancestors(polygon);
remaining_nodes = graph.nodes - polygon - ancestors;
if *len(remaining_nodes) == 0* **then**
  S_{left} = [] // leftover sites;
  for *site in sites* **do**
    if *site not in S_{loc}* **then**
      S_{left}.append(site);
  end
  return CS_{pre} + S_{loc}, S_{left};
CS = {} // sites lying in polygon;
for *site in S_{loc}* **do**
  if *site.location in polygon* **then**
    CS.add(site);
end
CS_{only} = {} // sites lying only in current polygon;
for *site in CS* **do**
  if *site.location not in remaining_nodes* **then**
    CS_{only}.add(site);
end
if *len(CS_{only}) > 0 or len(CS_{pre}) > 0* **then**
  return CS_{only} + CS_{pre}, sites - CS_{only} - CS_{pre};
if *len(CS) == 0* **then**
  return {}, sites;
site = sites.pop();
sites = sites - site;
return {site}, sites;

Algorithm 7. *divide*

Data: CP - polygon, V - vertices, S - sites, G - graph
Result: G_i – subgraphs
fsi = find_first_site_index(S, V[1..]) + 1;
fhi = max(1, fsi) // First head index;
lines = [Line(V_0, V_i) for i in fhi..];
// $PredPoly$ on the right side of the line l;
P_l^r = [$G.P_l^r(CP$, line) for line in lines];
heads = V_{fhi}.. ;
// Alg. 8: ;
S_r, R = right_sites_and_requirements(heads, S);
for *i in 0..* **do**
  if *$P_{l\ i}^r$.area >= R_i* **then**
    break;
end
if *P_{li}^r.area >= R_i* **then**
  if *i == fsi* **then**
    lines = [Line(V_j, V_i) for j in i..1];
    P_l^r = [$G.P_l^r(CP$, line) for line in lines];
    for *j in i..1* **do**
      if *$P_{l\ j}^r$.area >= R_i* **then**
        break;
    end
  pi = i == fsi ? i : 0 // Pivot index;
  li = i == fsi ? j + 1 : i - 1 // Low area index;
  hi == i == fsi ? j : i // High area index;
  T = Polygon(V_{pi}, V_{li}, V_{hi});
  P_l^r = $G.P_l^r(CP$, line);
  P_l^l = $G.P_l^l(CP$, line);
  edge = pi == 0 ? Line(V_{li}, V_{hi}) : Line(V_{hi}, V_{li});
  if *P_l^r.area + T.area > R_i* **then**
    R' = R_i - P_l^r.area;
    p = interpolate(R', V_{pi}, V_{li}, V_{hi});
    T' = Polygon(V_{li}, p, V_{pi});
    G_0 = G.subgraph(P_l^r, CP, $P_l^r + T'$, S_r);
    G_1 = G.subgraph(P_l^l, CP, $P_l^l - T'$, S - S_r);
    return G_0, G_1;
  P_e = $G.P_l^r(CP$, edge);
  if *P_l^r.area + P_e.area < R_i* **then**
    R' = R_i - P_l^r.area + P_e.area;
    p = interpolate(R', V_{pi}, V_{li}, V_{hi});
    T' = Polygon(V_{li}, p, V_{pi});
    G_0 = G.sub(P_l^r + P_e, CP, $P_l^r + T'$, S_r);
    G_1 = G.sub(P_l^l - P_e, CP, $P_l^l - T'$, S - S_r);
    return G_0, G_1;
  else
    T' = Polygon(V_{li}, midpoint(V_{li}, V_{hi}), V_{pi});
    G_0 = G.sub(P_l^r, CP, $P_l^r + T'$, S_r);
    G_1 = G.only_nodes_from(P_e);
    G_2 = G.sub(P_l^l - P_e, CP, $P_l^l - T'$, S - S_r);
    return G_0, G_1, G_2;
  end
else
  line = Line(midpoint(V_{-1}, V_0), V_{fsi});
  P_l^r = $G.P_l^r(CP$, line);
  P_l^l = $G.P_l^l(CP$, line);
  G_0 = G.sub(P_l^l, CP, P_l^l, S_fsi);
  G_1 = G.sub(P_l^r, CP, P_l^r, S - S_{fsi});
  return G_0, G_1;
end

Algorithm 8. right_sites_and_requirements

Data: V - vertices, sites
Result: S_r - right sites, R - requirements
S_r = [];
R = [];
// Mapping of sites' locations to sites;
SR = {site.location $\rightarrow$ site for site in sites};
first_site = $SR[V_0]$;
S_r.append({first_site});
R.append(first_site.requirement);
S_Σ = {} // set of accumulated sites;
R_Σ = 0 // accumulated requirements;
for V_i **in** $V[..-1]$ **do**
  if V_i **in** SR **then**
    S_i = $SR[V_i]$;
    S_Σ.add(S_i);
    R_Σ += S_i.requirement;
  S_r.append(S_Σ);
  R.append(R_Σ);
end
return S_r, R;

$$w = 2z\tan\gamma\left[\sin\alpha + \cos\alpha\tan\left(\frac{\pi}{2} - \alpha - \beta\right)\right] \quad (1)$$

where z is the altitude of the UAV, β and γ are half of the horizontal and vertical field-of-view, respectively, and α determines the tilt of the camera.

Metrics

There are several ways we can measure the quality of the obtained subpolygons and the trajectories assigned. We propose the following metrics:

i. Flight time
ii. Number of turns
iii. Useful path
iv. Compactness

The **flight time** can be greater for those partitions, where UAVs have to fly outside of the area or where trajectory self-intersects due to the specific configuration of the area. This metric is also related to the **number of turns**, which is especially relevant for fixed-wing vehicles. The more turns

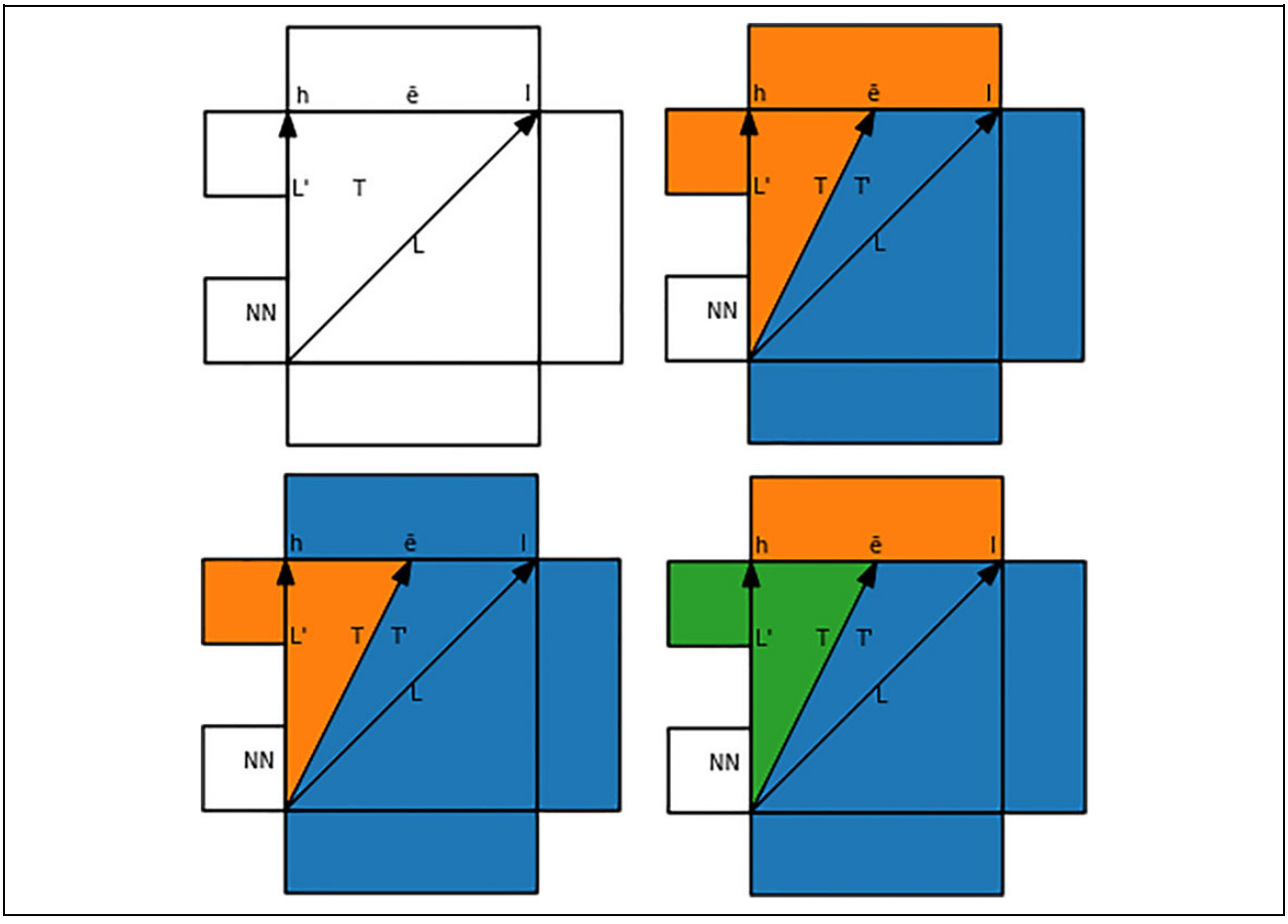

Figure 3. (a) Three subcases of case 4. L: penultimate position of the line splitter that is being rotated counter-clockwise; L′: the last position of the line splitter; T: triangle formed by L and L′; l: vertex corresponding to the lowest area; h: vertex corresponding to the highest area; e: the edge where a point is found for a head of a new line splitter; T′: triangle that corresponds to the new line-splitter position; NN: next neighbor. (b–d) Resulting subgraphs are highlighted by different colors.

Algorithm 9. *prepend*

Data: G - graph, SG - subgraphs, T - target node (optional)
Result: G' - updated graph
G' = EmptyGraph();
nodes = [];
edges = [];
for SG_i **in** SG **do**
  for *node* **in** SG_i **do**
    nodes.append(node);
  end
  for *edge* **in** $SG_i.edges$ **do**
    edges.append(edge);
  end
end
G'.add_nodes_from(nodes + G.nodes);
if *len(SG) == 2* **then**
  G'.add_edges_from(edges + G.edges);
  if T *is not None* **then**
    for N **in** *nodes* **do**
      $I = N \cap T$ // intersection;
      if I *is Segment* **then**
        G'.add_edge(N, T, I);
    end
else
  $A, B, C = SG$;
  for N **in** B **do**
    $I = N \cap A_0$;
    if I *is Segment* **then**
      edges.append(Edge(A_0, N, I));
    $I = N \cap C_{-1}$;
    if I *is Segment* **then**
      edges.append(Edge(N, C_{-1}, I));
  end
  G'.add_edges_from(edges + G.edges);
  if T *is not None* **then**
    $I = C_{-1} \cap T$;
    if I *is Segment* **then**
      G'.add_edge(C_{-1}, T, I);
end
for *graph* **in** *[G]* + SG **do**
  for *node* **in** *graph* **do**
    G'[node].sites = graph[node].sites;
  end
end
return G';

a UAV has to perform, the greater the time it will spend on them.

The percentage of **useful path** takes into account the area partitions that result in trajectories with significant sections lying outside of the corresponding polygons, as it will be shown below. We define it as follows

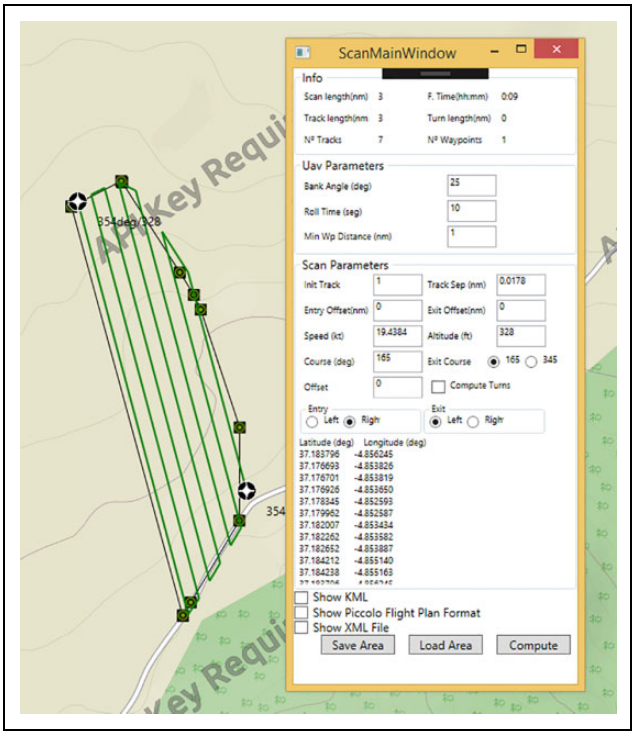

Figure 4. Screenshot of an application that calculates trajectories for a given polygon.

$$\text{Useful path} = \frac{\text{Path inside polygon}}{\text{Total path}} \times 100\%$$

Finally, the compactness of a polygon, also known as the shape index, is a numerical value that represents the degree to which the shape is similar to a circle. Several ways of calculating compactness can be found in the literature.[13] In this work, the **compactness** is defined as

$$\text{Compactness} = \frac{\text{Area (Polygon)}}{\text{Perimeter (Polygon)}}$$

Results

This section presents the metrics results for a number of randomly generated polygons. Then, it shows the partition details for a real case.

Extensive evaluation

The multiflight planner algorithm has been extensively evaluated for four different approaches. We named them as A–D. The A partition is based on Delaunay triangulation and the initial positions of the UAVs are fixed. In B, the initial positions are also fixed, but triangles from Delaunay triangulation are joined together to decrease the total number of convex parts. In C, the initial locations of the UAVs are not given and the initial partition was done by Delaunay triangulation. Finally, D is same as C but with triangles

Table 1. Algorithm approaches.

Approach	Triangulation	UAV start
A	Delaunay	Fixed
B	Joined	Fixed
C	Delaunay	Flexible
D	Joined	Flexible

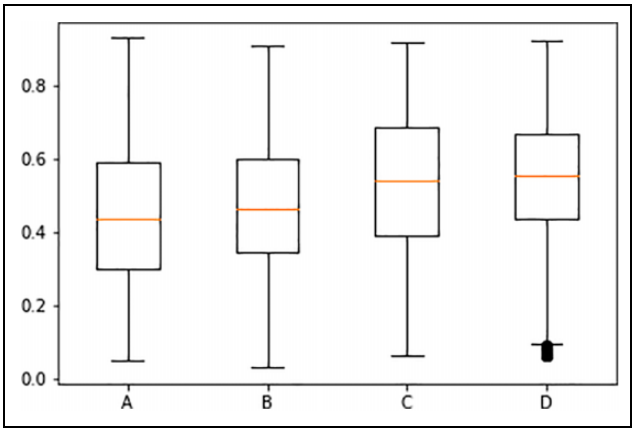

Figure 5. Compactness for four different approaches, A–D, and 100 randomly generated polygons.

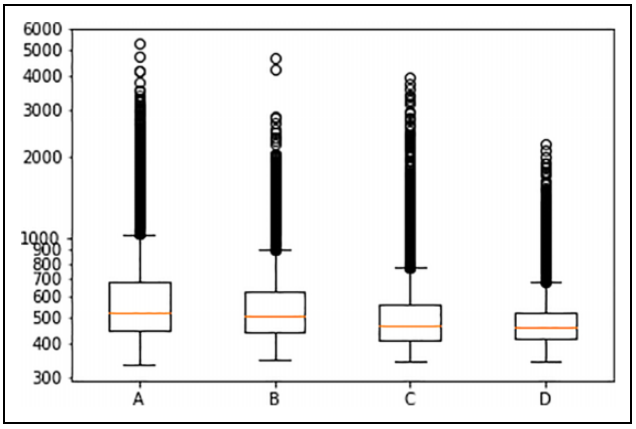

Figure 6. Time of flight for four different approaches, A–D, and 100 randomly generated polygons.

joined together to decrease the number of convex parts. Table 1 presents the summary of the four evaluated variants of the algorithm.

The extensive execution includes 100 polygons with sizes ranging from 3 to 50 vertices, with or without holes, and with the number of UAVs in the team ranging from 2 to 10, all having the same area split requirement. Times are calculated considering the speed of the UAVs equal to 10 m/s. Figures 5 to 8 show the global results of the four metrics presented above. It can be observed that the compactness of the four approaches (Figure 5) is between 0.3 and 0.6, having still some margin for improvement. Cases A and C exhibit higher

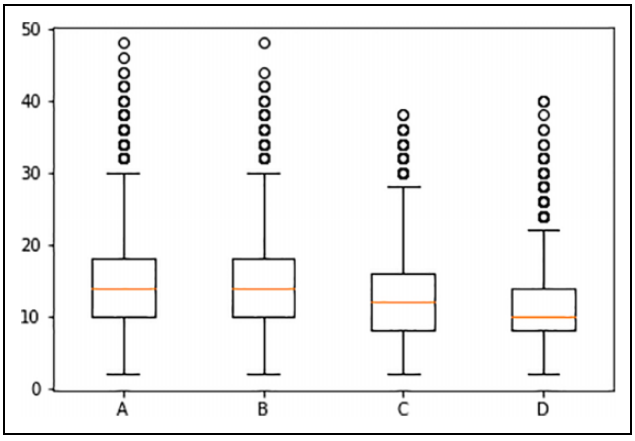

Figure 7. Number of turns for four different approaches, A–D, and 100 randomly generated polygons.

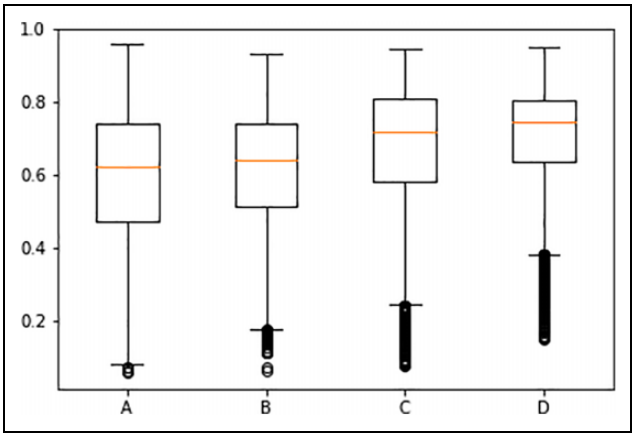

Figure 8. Percentage of useful path for four different approaches, A–D, and 100 randomly generated polygons.

variability, and cases C and D exhibit higher mean, being approach D the best of the four. Similar conclusion can be extracted from the other three figures (Figures 6 to 8), which ratify our assumption that compactness is a good metric to anticipate the quality of the flight trajectories.

For each metric, we also show the statistics regarding the number of UAVs (Figures 9 to 12). Each horizontal line shows the average value of the metric, and the vertical lines give the standard deviation of all four approaches, each one in a different color. As can be seen from the figures, joining triangles obtained from Delaunay triangulation (approaches B and D) has a positive effect on the resulting partitions, increasing the compactness and reducing the time of flight and the number of turns. Hence, we can assume that having larger convex parts will result in a more compact partition and a shorter time of flight. It can also be seen that relaxing the initial positions of the UAVs results in more compact subpolygons and shorter flight times. Notice from Figure 10 that the flight time is almost constant in approach D across the different number of UAVs. Since

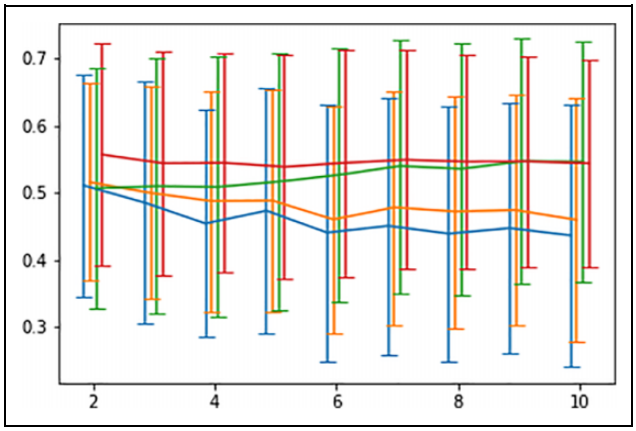

Figure 9. Compactness versus number of UAVs. Blue—case A, orange—case B, green—case C, red—case D. UAV: unmanned aerial vehicle.

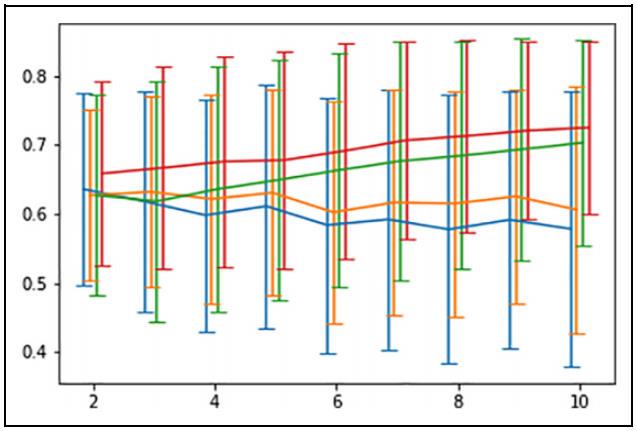

Figure 12. Useful path percentage versus number of UAVs. Blue—case A, orange—case B, green—case C, red—case D. UAV: unmanned aerial vehicle.

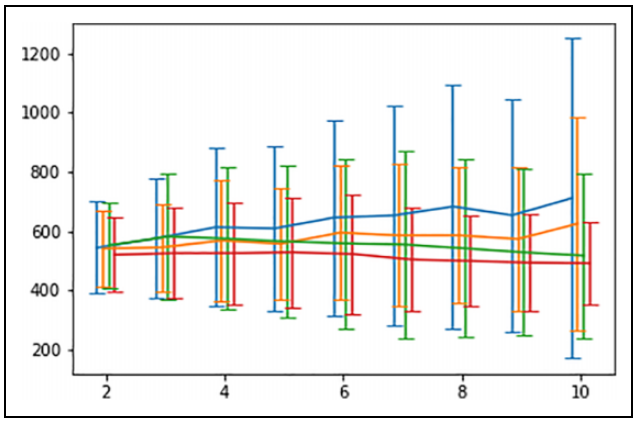

Figure 10. Time versus number of UAVs. Blue—case A, orange—case B, green—case C, red—case D. UAV: unmanned aerial vehicle.

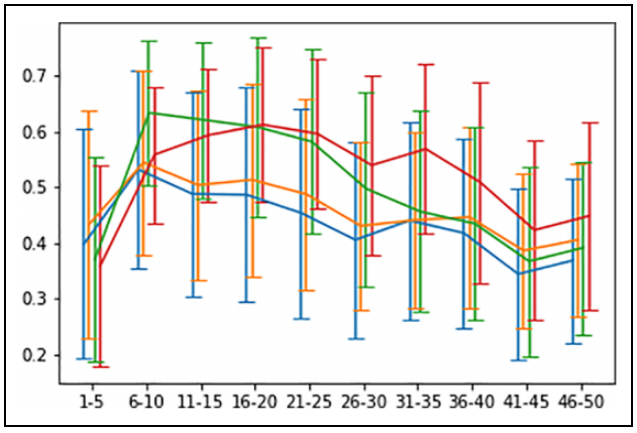

Figure 13. Compactness versus number of vertices of the polygon. Blue—case A, orange—case B, green—case C, red—case D.

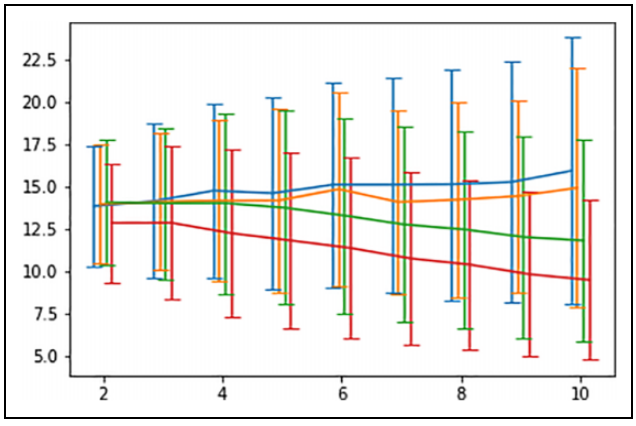

Figure 11. Turns versus number of UAVs. Blue—case A, orange—case B, green—case C, red—case D. UAV: unmanned aerial vehicle.

the number of UAVs is directly proportional to the area of the polygon, we can conclude that approach D scales perfectly with the area.

Finally, Figure 13 shows the evolution of the compactness in relation to the number of vertices of the polygon to split. We can observe that approach D is not always the best approach but only for polygons with more than 20 vertices. For smaller polygons, approaches B and C can obtain more compact partitions. However, it seems logical to think that in a real situation, the number of vertices will be greater than a couple of tens, and thus, approach D would be the most convenient.

Case of study

As a case of study, we would like to obtain a partition of a real parcel for two, three, and four UAVs with equal area requirements and assign the trajectories. Figure 14 shows the initial area of the parcel and its partition into convex parts by triangulation. It can be seen that the number of subpolygons is too high. This can be fixed by removing vertices of the polygon that are too close to each other. The result of such a transformation is shown in Figure 15. In our

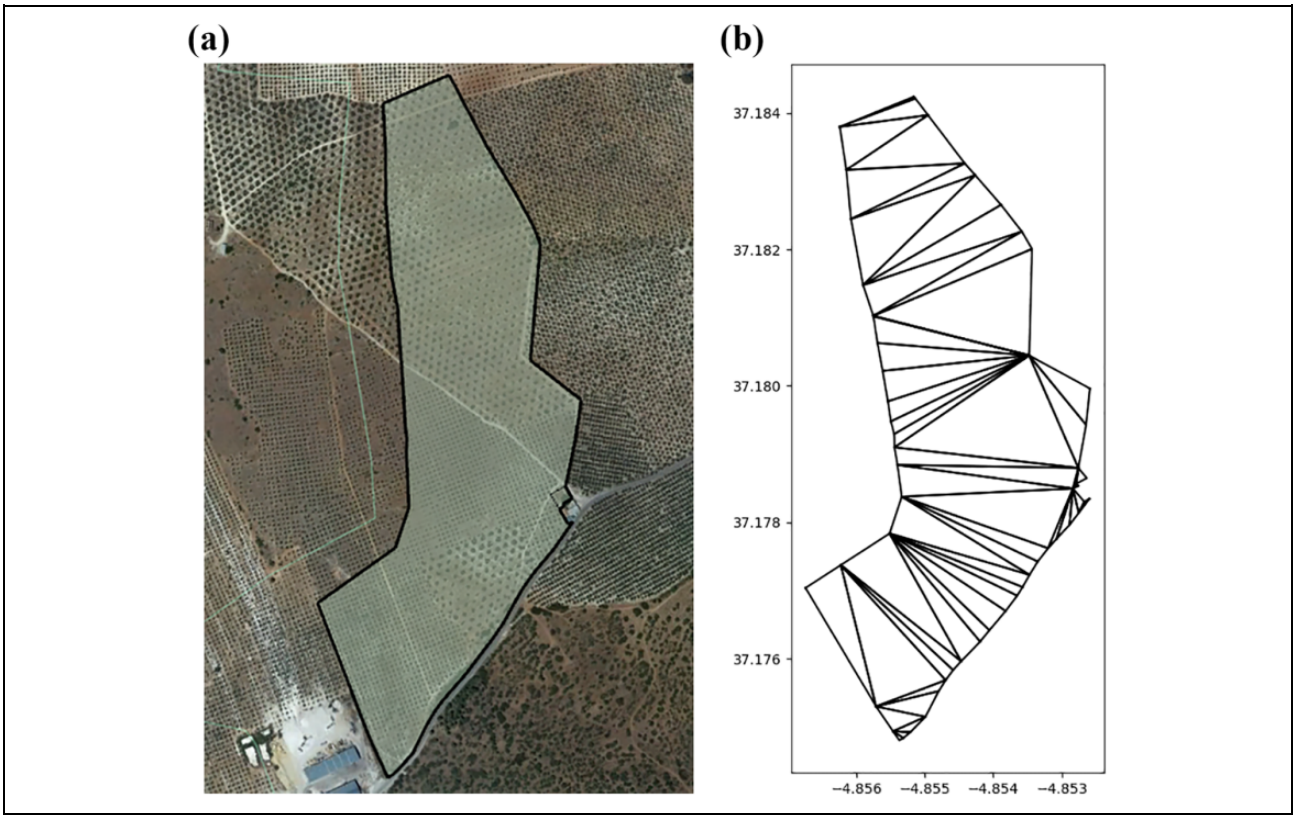

Figure 14. (a) Aerial view of the parcel used as case of study and (b) its partition into convex polygons by triangulation.

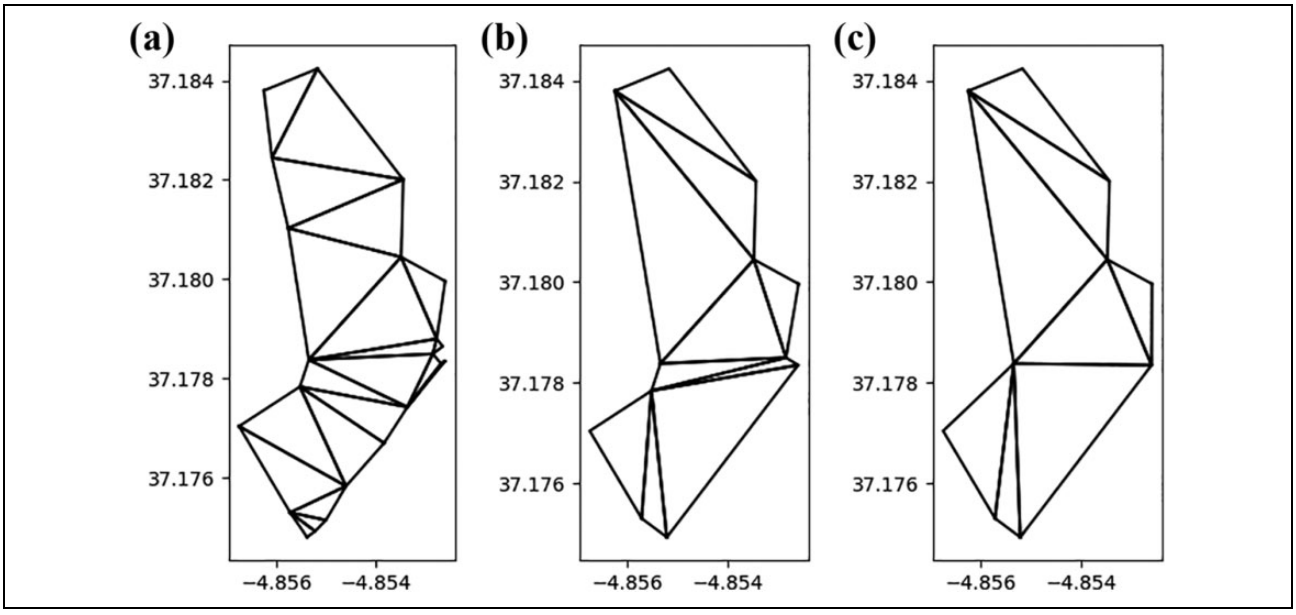

Figure 15. (a–c) Partition of the original parcel with reduced number of points to convex polygons by triangulation. Results for three different levels of details are presented.

case, we chose the desired partition shown in the middle of the figure, but, generally, the level of resolution of the border depends solely on the mission characteristics.

First, we split the area into two to four convex parts corresponding to the number of UAVs according to the algorithm mentioned in the previous section. The resulting partition along with starting positions of the UAVs can be seen in Figure 16.

Finally, those subareas are used to compute the trajectories of each UAV. To calculate the distance between

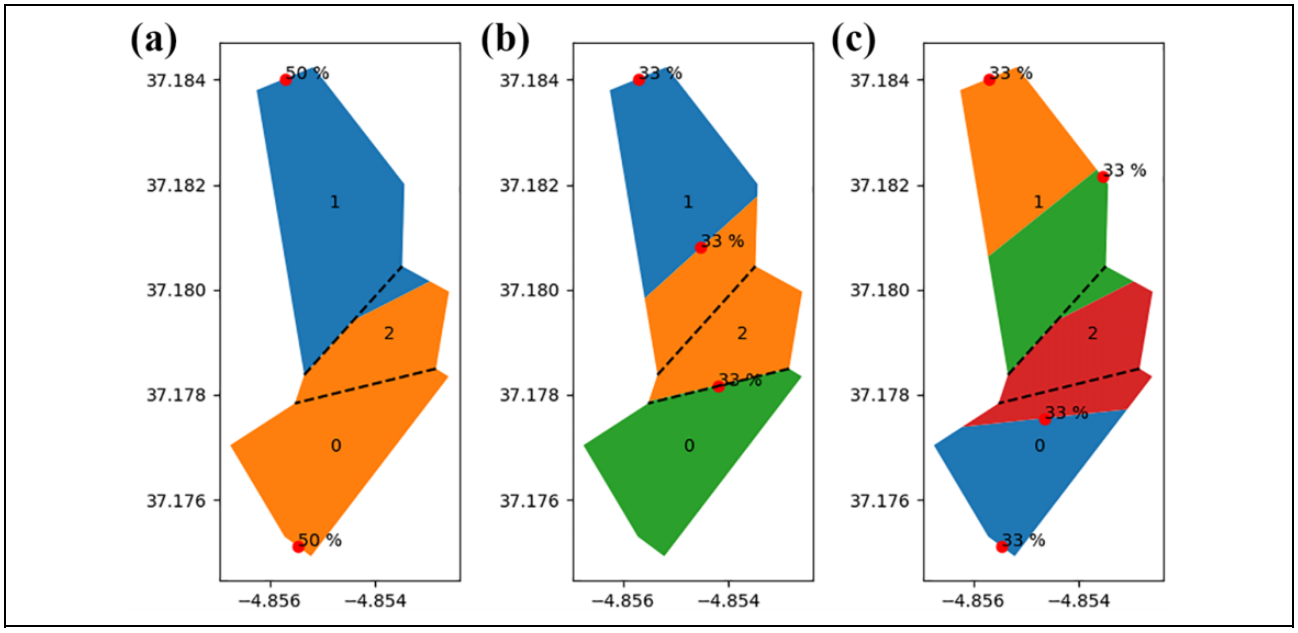

Figure 16. (a–c) Area partition for two, three, and four UAVs. Initial positions are marked by dots. Dashed lines show convex parts obtained from joining triangles of Delaunay triangulation. UAV: unmanned aerial vehicle.

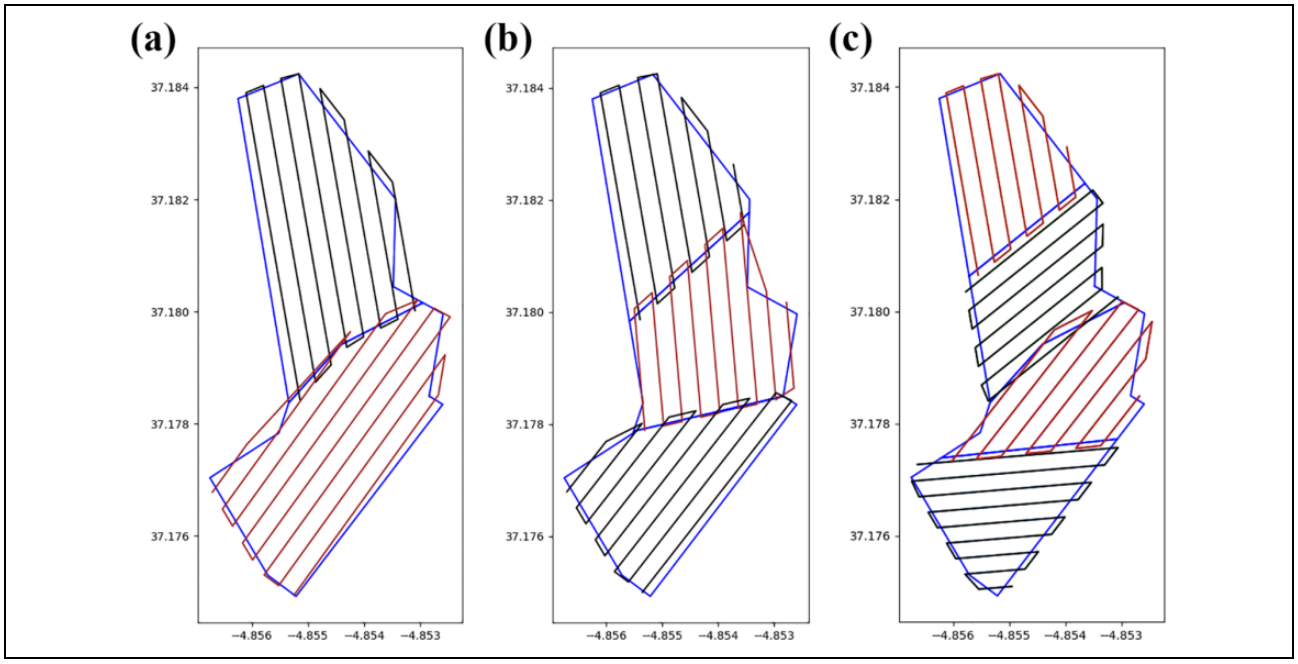

Figure 17. (a–c) Area partition for two, three, and four UAVs along with assigned trajectories for each UAV. UAV: unmanned aerial vehicle.

tracks, the formula (1) was used to calculate the sensing width. In our case, altitude was set to 100 m, camera tilt to 0°, and horizontal field-of-view of the UAV's camera to 79°. This gives a sensing width of 165 m. For an 80% image overlapping, the track separation distance is 33 m. The obtained tracks are shown in Figure 17. It can be seen that there is some overlapping between the areas assigned to different UAVs. This can lead to some areas photographed more than once.

Trajectories for the case of four UAVs and for two approaches using Delaunay triangulation and joined triangles are calculated according to the description in the previous section. The results are depicted in Figure 18. As can be seen, in this case, having larger convex parts have a positive effect on both the final partition and the assigned trajectories. Due to irregular shapes of the polygons generated based on the partition with Delaunay

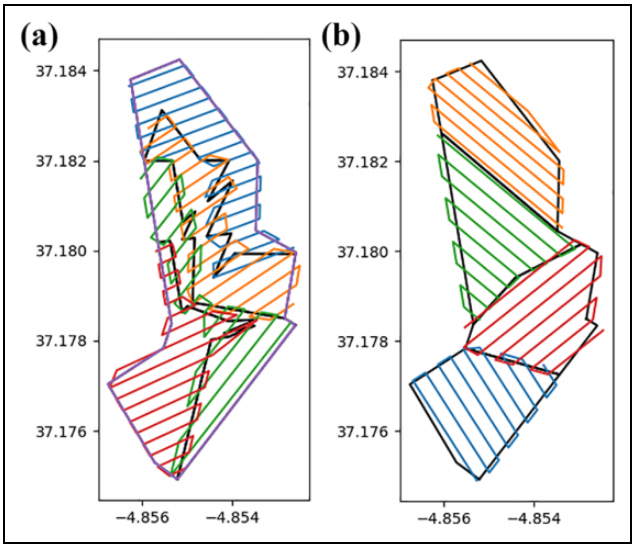

Figure 18. (a, b) Trajectories for four UAVs calculated for two ways to generate partitions: based on partition from constrained Delaunay triangulation and when some of the resulting triangles are joined together to form larger parts. UAV: unmanned aerial vehicle.

triangulation, the percentage of the useful path becomes significantly lower.

Conclusions

In this work, we implemented a flight planner that divides an area of interest into several subareas and assigns each of them to a UAV. The UAVs follow a specific trajectory that is built according to the input parameters and given constraints. The main contribution of this work is the first open-source implementation of the algorithm by Hert and Lumelsky[5] that is able to split any complex polygon. While there were a couple of papers that tried to improve that algorithm, none of them actually met our requirements. For example, in one paper, Bast and Hert[14] removed the initial positions of robots completely from their algorithm, and in another one,[15] the algorithm was adopted to solve a dynamic partitioning problem instead of a static one. Above the original algorithm, several extensions have been included, such as how sites should be assigned to the convex part of an input polygon, even when some or all UAVs do not have specified initial positions.

A metrics analysis was run for a sample of 100 randomly generated polygons, which showed that partitioning the input polygon into larger convex pieces helped to obtain more compact resulting parts and improved the quality of the trajectories generated over them. Currently, there is a lack of algorithm implementations for partitioning polygons into convex parts. In the future, an algorithm to produce the minimum number of convex parts[16] should be implemented. Trapezoidal partitioning[17] could be also incorporated into our algorithm. We assume that it can produce parts that are especially appropriate for the back-and-forth sweeping pattern. In fact, we consider that any partitioning process to convex parts should happen at the same time as creating a region-adjacency graph to reduce run-time but, to the best of our knowledge, there are no works done in this area.

The part of trajectory generation should also be improved. Currently, the trajectories of several UAVs can get overlapped between each other, which is not desirable since they will spend more time and there will be a risk of a mid-air collision. Also, right now, the sweeping happens only in one direction per each subpolygon but changing them could be beneficial, especially for non-convex areas.

Currently, the trajectories of several UAVs can overlap each other. This overlapping is undesirable as the UAVs will fly longer and there will be a risk of a mid-air collision. In this sense, the generation of trajectories can be improved by employing several techniques. One of the simplest solutions is to restrict the UAVs from crossing the boundaries between their corresponding subpolygons. However, this will not work in all cases. One can imagine a case when a subpolygon has a narrow part enclosed by neighboring subpolygons. This narrow part may be inaccessible for a UAV sweeping in a parallel manner. The problem of inaccessibility together with overlapping can be probably addressed by sweeping in several directions for each subpolygon. This, however, is a more complex task, and we leave its analysis for future work.

Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work was supported by Ministerio de Economía, Industria y Competitividad, and Gobierno de España under grants/award numbers BES-2017-079798 and TRA2016-77012-R.

ORCID iDs

Georgy Skorobogatov https://orcid.org/0000-0003-2536-1470
Cristina Barrado https://orcid.org/0000-0003-0100-724X
Esther Salamí https://orcid.org/0000-0002-4635-2963

References

1. Skorobogatov G, Barrado C, and Salamí E. "Multiple UAV systems: A survey." *Unmanned Syst* 2020; 8(2): 149–169.
2. Royo P, Perez-Batlle M, Cuadrado R, et al. "Enabling dynamic parametric scans for unmanned aircraft system remote sensing missions." *J Aircraft* 2014; 51: 870–882.
3. Cabreira T, Brisolara L, and R Ferreira P. "Survey on coverage path planning with unmanned aerial vehicles." *Drones* 2019; 3: 4.

4. Maza I and Ollero A. "Multiple UAV cooperative searching operation using polygon area decomposition and efficient coverage algorithms." *Distributed Auton Robot Syst* 2007; 6: 221–230.
5. Hert S and Lumelsky V. "Polygon area decomposition for multiple-robot workspace division." *Int J Comput Geom Appl* 1998; 8: 437–466.
6. Barrientos A, Colorado J, Cerro JD, et al. "Aerial remote sensing in agriculture: a practical approach to area coverage and path planning for fleets of mini aerial robots." *J Field Robot* 2011; 28: 667–689.
7. Valente J, Cerro JD, Barrientos A, et al. "Aerial coverage optimization in precision agriculture management: a musical harmony inspired approach." *Comput Electron Agric* 2013; 99:153–159.
8. Berger C, Wzorek M, Kvarnström J, et al. "Area coverage with heterogeneous uavs using scan patterns." In: *2016 IEEE international symposium on safety, security, and rescue robotics (SSRR)*, Lausanne, Switzerland, 23–27 October 2016, pp. 342–349. Lausanne: IEEE.
9. Greene DH. "The decomposition of polygons into convex parts." *Comput Geom* 1983; 1: 235–259.
10. De Berg M, Van Kreveld M, Overmars M, et al. *Computational geometry*. Berlin, Heidelberg: Springer, 1997.
11. Hagberg A, Swart P, and S Chult D. *Exploring network structure, dynamics, and function using NetworkX*. Report No. LA-UR-08-05495; LA-UR-08-5495, 2008, Los Alamos, NM: Los Alamos National Lab (LANL).
12. Bondy JA and Murty US. *"Graph theory with applications" Grad. Texts in Math*. London: Macmillan, 1976.
13. Li W, Goodchild MF, and Church R. "An efficient measure of compactness for two-dimensional shapes and its application in regionalization problems." *Int J Geogr Inf Sci* 2013; 27(6): 1227–1250.
14. Bast H and Hert S. "The area partitioning problem." In: *12th Canadian Conference on Computational Geometry*, Fredericton, New Brunswick, 2000, pp. 163–172.
15. Hert S and Richards B. "Multiple-robot motion planning= parallel processing+ geometry." In: Hager GD, Christensen HI, Bunke H, et al. (eds) *Sensor based intelligent robots*, Berlin: Springer, 2002, pp. 195–215.
16. Chazelle B and Dobkin D, "Decomposing a polygon into its convex parts." In: *Proceedings of the eleventh annual ACM symposium on theory of computing*, 30 April 1979, pp. 38–48. New York: ACM.
17. Lorenzetto GP, Datta A, and Thomas RC, "A fast trapezoidation technique for planar polygons." *Comput Graph* 2002; 26(2): 281–289.