Optimizing Workspace Division for Multi-UAV Systems

GEORGY SKOROBOGATOV
Aeronautical Engineer

Advisors
DR. CRISTINA BARRADO MUXÍ
DR. ESTHER SALAMÍ SAN JUAN

Doctorate program in Aerospace Science and Technology
Department of Computer Architecture - Aerospace Engineering Division
Technical University of Catalonia - BarcelonaTech

A dissertation submitted for the degree of
Doctor of Philosophy
July 2023

Optimizing Workspace Division for Multi-UAV Systems

Author
Georgy Skorobogatov

Advisors
Dr. Cristina Barrado Muxí
Dr. Esther Salamí San Juan

Reviewers
Ismael Colomina Fosch
Daniel Delahaye

Thesis committee
Ismael Colomina
Daniel Delahaye
Enric Pastor

Doctorate program in Aerospace Science and Technology
Technical University of Catalonia - BarcelonaTech
July 2023

PhD. Thesis made in:
Department of Computer Architecture - Aerospace Engineering Division
Esteve Terradas, 5
08860 Castelldefels (Barcelona), Spain

Contents

List of Figures

Figures in Appendices

List of Tables

List of Algorithms

List of Listings

List of Publications

The list of publications resulting from this PhD work is given in inverse chronological order as follows:

Journal Papers

- SKOROBOGATOV, GEORGY, BARRADO, CRISTINA & SALAMÍ, ESTHER. 2023. Multi-robot workspace division based on Bottom-up algorithm for polygon decomposition. *(in preparation)*.
- SKOROBOGATOV, GEORGY, BARRADO, CRISTINA & SALAMÍ, ESTHER. 2021. Multi-robot workspace division based on compact polygon decomposition. *IEEE Access*. D.O.I.: 10.1109/ACCESS.2021.3134760. **9**, 165795–165805.
- SKOROBOGATOV, GEORGY, BARRADO, CRISTINA, SALAMÍ, ESTHER & PASTOR, ENRIC. 2021. Flight planning in multi-unmanned aerial vehicle systems: Nonconvex polygon area decomposition and trajectory assignment. *International Journal of Advanced Robotic Systems*. D.O.I.: 10.1177/1729881421989551. **18(1)**, 1729881421989551.
- SKOROBOGATOV, GEORGY, BARRADO, CRISTINA & SALAMÍ, ESTHER. 2020. Multiple UAV systems: A survey. *Unmanned Systems*. D.O.I.: 10.1142/S2301385020500090. **8(02)**, 149–169.

Abstract

With the advance of unmanned aerial vehicle (UAV)-related technology using several drones in the context of a single mission becomes more and more common. New problems and challenges appear as a result.

When analyzing research works on using systems of multiple UAVs we could notice that the majority of the authors provided few details on how the path planning or workspace division was done. Out of those researchers who mentioned it, some pointed out that the planning or area partitioning was performed by hand. Other researchers presented brief ideas of algorithms with too little information to implement ~~it~~ them. In other research works very brief lists of algorithms were given that could solve the problem. And even in those cases when the information on the algorithms was provided, the algorithms themselves did not have any freely available implementations.

The purpose of this thesis is to fill the gap in the area of workspace division in order to facilitate the usage of systems consisting of multiple UAVs.

In order to accomplish the aforementioned goal, in this thesis, we performed an analysis of the literature on the subject of workspace decomposition between multiple robots and UAVs in particular. As it will be shown later, there are almost no research works published in this area.

We implemented two ~~state-of-the-art~~ state-of-the-art algorithms and shared information on how we achieved that and what ~~were the~~ aspects ~~that~~ needed clarification or could be improved. We analyzed thoroughly the produced results, and propose improvements to the algorithm that yielded better results. And finally, we proposed, implemented, and analyzed two alternative algorithms based on the obtained experience. These algorithms outperformed the algorithm from the literature in terms of the quality of the resulting partition. One algorithm solves the partition problem for convex polygons and the other one solves the partition problem for non-convex polygons. Finally, we have summarized a set of open problems that could be solved in the future.

Resumen

Con el avance de la tecnología relacionada con UAV, el uso de varios drones en el contexto de una sola misión se vuelve cada vez más común. Como resultado, surgen nuevos problemas y desafíos.

Al analizar los trabajos de investigación sobre el uso de sistemas de múltiples UAVs pudimos notar que la mayoría de los autores proporcionaron pocos detalles sobre cómo se realizaba la planificación de caminos o la división del espacio de trabajo. De aquellos investigadores que lo mencionaron, algunos señalaron que la planificación o partición de áreas se realizaba a mano. Otros investigadores presentaron breves ideas de algoritmos con muy poca información para implementarlos. En otros trabajos de investigación se dieron listas muy breves de algoritmos que podrían resolver el problema. E incluso en aquellos casos en los que se proporcionó información sobre los algoritmos, los propios algoritmos no tenían implementaciones disponibles gratuitamente.

El propósito de esta tesis es llenar el vacío en el área de la división del espacio de trabajo para facilitar el uso de sistemas que consisten en múltiples UAVs.

Para lograr el objetivo antes mencionado, en esta tesis, realizamos un análisis de la literatura sobre el tema de la descomposición del espacio de trabajo entre múltiples robots y UAVs en particular. Como se verá más adelante, casi no hay trabajos de investigación publicados en esta área.

Implementamos dos algoritmos de última generación y compartimos información sobre cómo lo logramos y cuáles eran los aspectos que necesitaban aclaración o que podrían mejorarse. Analizamos a fondo los resultados producidos y proponemos mejoras al algoritmo que arrojaron mejores resultados. Y finalmente, propusimos, implementamos y analizamos dos algoritmos alternativos basados en la experiencia obtenida. Estos algoritmos superaron al algoritmo de la literatura en términos de calidad de la partición resultante. Un algoritmo resuelve el problema de partición para polígonos convexos y el otro resuelve el problema de partición para polígonos no convexos. Finalmente, hemos resumido un conjunto de problemas abiertos que podrían resolverse en el futuro.

Resum

Amb l'avenç de la tecnologia relacionada amb UAV, l'ús de diversos drones en el context d'una sola missió esdevé cada cop més comú. Com a resultat, sorgeixen nous problemes i desafiaments.

En analitzar els treballs de recerca sobre l'ús de sistemes de múltiples UAVs vam poder notar que la majoria dels autors van proporcionar pocs detalls sobre com es feia la planificació de camins o la divisió de l'espai de treball. D'aquells investigadors que ho van esmentar, alguns van assenyalar que la planificació o la partició d'àrees es feia a mà. Altres investigadors van presentar idees breus d'algorismes amb molt poca informació per implementar-los. En altres treballs de recerca es van fer llistes molt breus d'algorismes que podrien resoldre el problema. I fins i tot en aquells casos en què es va proporcionar informació sobre els algoritmes, els mateixos algoritmes no tenien implementacions disponibles gratuïtament.

L'objectiu d'aquesta tesi és omplir el buit a l'àrea de la divisió de l'espai de treball per facilitar l'ús de sistemes que consisteixen en múltiples UAVs.

Per assolir l'objectiu abans esmentat, en aquesta tesi fem una anàlisi de la literatura sobre el tema de la descomposició de l'espai de treball entre múltiples robots i UAVs en particular. Com veurem més endavant, gairebé no hi ha treballs de recerca publicats en aquesta àrea.

Implementem dos algorismes d'última generació i compartim informació sobre com ho aconseguim i quins eren els aspectes que necessitaven aclariment o que es podrien millorar. Analitzem a fons els resultats produïts i proposem millores a l'algorisme que van donar resultats millors. I finalment, vam proposar, implementar i analitzar dos algorismes alternatius basats en l'experiència obtinguda. Aquests algoritmes van superar l'algorisme de la literatura en termes de qualitat de la partició resultant. Un algorisme resol el problema de partició per a polígons convexos i l'altre resol el problema de partició per a polígons no convexos. Finalment, hem resumit un conjunt de problemes oberts que es podrien resoldre en el futur.

List of Acronyms

2D	two-dimensional
3D	three-dimensional
CPP	coverage path planning
NFZ	no-fly zone
RAG	region-adjacency graph
RPAS	remotely piloted aircraft system
UAS	unmanned aerial system
UAV	unmanned aerial vehicle
WKB	well-known binary
WKT	well-known text

Glossary

anchored area partition of a polygon $\mathcal{P}$ using a set of n points p_i — such *area partition* that $\forall i : p_i \in \mathcal{P}_i$

area partition of a polygon $\mathcal{P}$ — a set of n nonoverlapping polygons $\mathcal{P}_i$ where $n > 1$ such that $\bigcup_{i=1}^{n} \mathcal{P}_i = \mathcal{P}$.

bounding box — rectangle covering all the points of a given geometry and having a minimum possible size with each side collinear to one of the coordinate axis

convex polygon — a polygon intersection with which of any line results in not more than one line segment.

multipolygon — a collection of non-overlapping polygons

predecessor of a given node in a directed and ordered graph is a node that is found before the given node when iterating over the nodes in the order of the graph.

reflex angle — an angle greater than $180\,\mathrm{deg}$ and less than $360\,\mathrm{deg}$

region-adjacency graph — a graph whose nodes represent polygons and whose edges represent common borders between them

shoelace formula or in some sources also called shoelace method, shoelace algorithm, Gauss's area formula, and surveyor's formula — a mathematical algorithm to calculate the area of a simple polygon given the coordinates of its vertices

simple polygon — a polygon without holes and without self-intersections.

Steiner points — points that are originally not part of the given geometric object but are included in the process of solving an optimization problem in order to obtain better results. Named after Jakob Steiner, a Swiss mathematician.

well-known binary binary equivalent of the well-known text

well-known text markup language for representing geometry objects

workspace decomposition — a process or a result of dividing a space where a robot or multiple robots operate into two or more smaller parts.

I

Introduction

I.1 Motivation of this PhD

I.1.1 On UAVs and their applications

An unmanned aerial vehicle (UAV), frequently called a drone, as the name implies, is an aircraft without a human on board. UAVs can be a part of unmanned aerial system (UAS) that apart from the UAV also includes a communication system and the control system located on the ground. UAVs may be operated by a human, be fully autonomous, or have some degree of autopilot assistance (Fahlstrom *et al.*, 2022; Skorobogatov *et al.*, 2020).

UAVs have a wide range of applications such as surveying and mapping (Zahari *et al.*, 2021), construction and infrastructure inspection (Feretzakis & Badogiannis, 2021; Lebedev & Izhboldina, 2022), search and rescue (Zimroz *et al.*, 2021; Atif *et al.*, 2021), reconnaissance (Jayaweera & Hanoun, 2021; Gao *et al.*, 2021), precision agriculture (Radoglou-Grammatikis *et al.*, 2020), goods delivery (Marintseva *et al.*, 2021; Lv *et al.*, 2021), road traffic monitoring (Byun *et al.*, 2021; Gupta & Verma, 2021) and many others (Shakhatreh *et al.*, 2019).

I.1.2 Multi-UAV systems

In recent years there has been a growing number of UAV usage including the usage of multiple UAS as shown in Skorobogatov *et al.* (2020). One can see in Fig. I-1 that the number of research works covering experiments, simulations, and different concepts of multi-UAV missions has increased significantly since the year 2013.

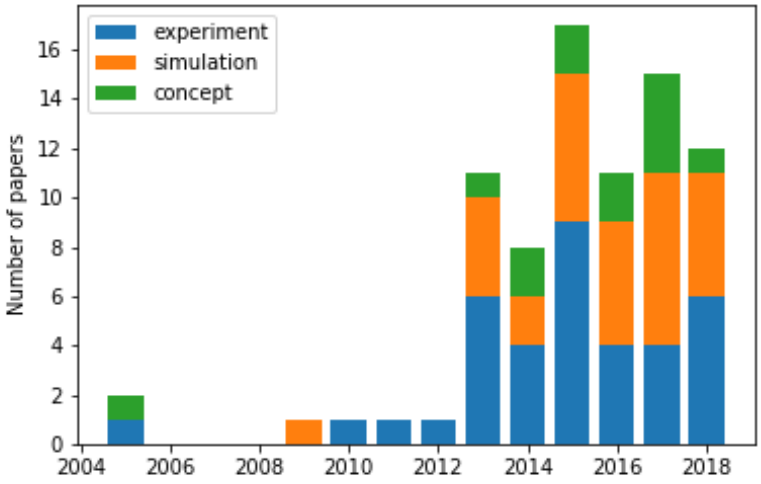


Figure I-1: *Yearly distribution of 65 publications discussing multiple UAVs covered in Skorobogatov* et al. *(2020)*

I.1.2.1 Advantages of multiple UAS

The reason for this growth in recent years lies in the numerous advantages of using such systems. Some of these advantages are discussed in Maza *et al.* (2015) and Chung *et al.* (2018), but after having analyzed a number of papers about multiple UAVs, we can extend the advantages to the following:

- **Time efficiency:** The mission operational time can be significantly reduced with the use of multiple UAVs. The most drastic effect can be found in tasks such as target search, exploration, etc. One of the examples is Han *et al.* (2013) addressing urgent detection of nuclear radiation in a disaster and cooperatively building a contour map before deploying the salvage. For another quantitative example, we could imagine a 50 ha field that is covered by a single UAV in up to 15 minutes and a natural park of 13,000 ha. To cover the area of this park with only one UAV it would require up to 64 hours excluding the time spent on recharging. Introducing a team of 20 UAVs for this task would significantly reduce that time to only 3 hours.

- **Cost:** Sometimes having a single operating UAV could be an expensive solution when having several small UAVs could be much cheaper and reduce the costs related to, for example, power consumption. One could imagine a use case with goods deliveries. Having a team of small UAVs will be a cheaper alternative to using a heavy ($> 25kg$) UAV due to the fact that to use this type of UAVs one has to go through long and costly administrative procedures with the final price being magnitudes higher.

- **Simultaneous actions:** A team of UAVs can accomplish tasks in different geographical locations at the same time contrary to a single UAV. This can be used when it is necessary to collect information from the points that cannot be reached by a single UAV. This is the case for the problems related to continuous coverage.

- **Complementarity:** In a team of UAVs each member can have a specific set of sensors. All the sets would be complementary to each other. This separation could be done when all the payload could not be physically located on a single UAV. Or when the mission goals demand different types of sensors to be located in different areas. This, for example, was done in one

of the experiments discussed in Merino *et al.* (2015) where one UAV was equipped with a fire detector and the second one with an infrared camera.

- **Fault tolerance:** In the case of a single UAV, the loss of it means a termination of the mission. But when there are multiple operating UAVs, the loss of a UAV could be mitigated by the algorithm managing the flight by assigning additional tasks to other UAVs. For example, in Scherer *et al.* (2015) one of the challenges for their system's architecture was to be fault tolerant. A search and rescue mission cannot be canceled if one of the UAVs in a team malfunctions.
- **Flexibility:** single UAV can perform one task at a time, while a group of UAVs could be dynamically allocated to different tasks at the same time and rearranged if necessary. As an example, one could imagine a case of a group of UAVs performing an observation on a crowd of people. If at some point there is a subgroup of people that separates from the main group, and it is required to track their movement, the flexible task allocating system would be able to rearrange the tasks and send one or more UAVs for tracking that subgroup.

I.1.2.2 Disadvantages of multiple UAS

As a result of this growth, new tasks and challenges are appearing. The disadvantages of using these systems can be summarized as follows:

- **Legal restrictions:** using several UAVs at the same time may be not permitted in some jurisdictions. For example, Shakhatreh *et al.* (2018) reports that in the United States, simultaneous use of large numbers of autonomous UAVs for commercial applications is not allowed by the Federal Aviation Administration.
- **Piloting complexity:** operating multiple UAVs can be challenging for a single pilot. As the number of UAVs in a system increases, it becomes more difficult to manage them. To facilitate the process of operating multiple UAVs, there is a need for advanced systems that can automate some of the tasks involved.
- **Safety issues:** introducing several UAVs presents several issues related to safety. One of them is collision avoidance which is also related to the piloting problem. With several vehicles in the air, it is necessary that they do not go into each others' buffer zones, otherwise, they can collide and crash on the ground. Also, since several UAVs cover a larger area than one vehicle, it becomes more difficult to avoid them in the air. There are many other things to consider regarding safety such as no-fly zones, flying in dense urban areas, malfunctioning, and the possibility of hacking. And the more complex the system is, the more difficult it is to account for all possible system failures.

I.1.3 On growing piloting complexity of multi-UAV systems

Operating multiple UAVs presents a number of challenges, including the growing complexity of piloting. Piloting multiple UAVs requires careful coordination and planning to avoid collisions and ensure that each vehicle is performing its intended mission. The survey by Skorobogatov *et al.* (2020) provides insights into how piloting is performed with multiple UAVs.

It is reported that in more than 70% of research papers path planning was performed before the launch of the UAVs, whereas in the remaining 30% of research papers, the UAVs were operated by users in real-time. The pre-planning can be performed in various ways. In some research works, the planning was done by specifying areas to be observed on a digital map (Yahyanejad & Rinner, 2015; Hattenberger *et al.*, 2014) or by specifying no-fly zones (NFZs) (Chen *et al.*, 2013). In other research works, specially designed mission intent languages were used (Maza *et al.*, 2011;

Bailon-Ruiz *et al.*, 2017; Muñoz-Morera *et al.*, 2016; Maza *et al.*, 2014; Torres-González *et al.*, 2017). Both ways, in fact, have the same underlying implementation as the information about checkpoints on the map gets translated into those languages. Out of those 70% of works with mission pre-planning more than 40% of research works reported that they performed the so-called "fixed algorithm assignment". The fixed algorithm assignment implies that at any point in time, it is possible to determine the exact location of a UAV.

With the growth of usage of multiple-UAV systems, new solutions will be required to facilitate their planning. For example, the goal of the FLOR project (Barrado *et al.*, 2016) is to redefine how pilots operate remotely piloted aircraft system (RPAS). If previously a dedicated crew had to manage a single aircraft, this project proposes that a fleet of RPAS be operated by a team of remote pilots. In this paradigm, each pilot would operate multiple aircraft during certain phases of flight, and the responsibility would be transferred between the pilots as the operation advances. The pilots then could be allocated to those tasks that fit better their capabilities and the tasks could be dynamically adjusted over time. It is argued that this will lead to improved economic efficiency and safety. Pilots would be able to supervise multiple RPAS when it is not required of them to perform many actions and could focus on individual aircraft during the critical flight stages.

I.1.4 Workspace division problem

In many practical cases, when a mission requires several UAVs to be deployed, the workspace has to be divided between them. Therefore, one of the tasks that have to be solved is workspace division.

The workspace decomposition problem consists of the representation of the workspace using such geometric objects as polygons or collections of polygons called multipolygons. In this thesis, we do not consider disconnected workspaces represented by multipolygons. Polygons, however, can contain holes that represent no-fly zones or obstacles. We do not consider nested polygons where holes can contain other polygons inside.

The resulting decomposition of the input polygon will contain parts of various areas. These areas are normally selected to represent the relative capabilities of UAVs and defined by so-called area requirements.

The initial positions of UAVs can also define the resulting decomposition in the sense that the points representing those locations should be included in the resulting areas corresponding to the UAV.

Currently, there is a very small number of research works that discuss workspace division, if we do not take into account works specifically about coverage path planning (CPP). Also, there are no freely available tools that are capable of performing the polygon decomposition according to the relative capabilities of several UAVs. In the following years, the usage of multiple UAVs simultaneously will be even more common. Quick analysis at dimensions.ai (Hook *et al.*, 2018) shown in Fig. I-2 demonstrates that the number of research works mentioning the phrase "multiple UAV" grew up more than twice from the year 2018 to the year 2022. Therefore, work has to be done in the area of workspace division to facilitate the usage of multiple UAV systems.

In the majority of the analyzed research works where multiple UAVs were deployed, the workspace decomposition was either performed manually or it was a by-product of CPP with generated trajectories covering parts of the area.

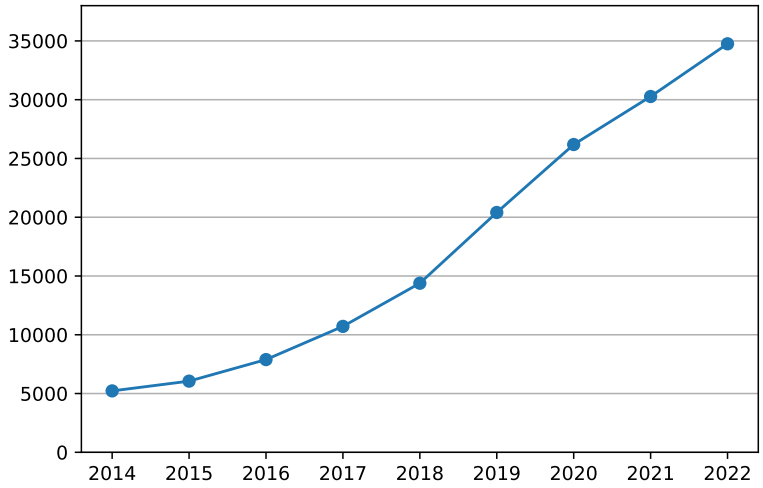


Figure I-2: *Number of publications per year mentioning "multiple UAV" in their texts according to dimensions.ai.*

I.2 Objectives of this PhD Thesis

In research works that talk about using systems consisting of multiple UAVs the problem of workspace division is barely discussed. Authors either send a reader elsewhere to read about it, implement an algorithm but do not share it or in many cases do not mention the problem at all. The main objective of this thesis is to fill the gap by providing algorithms and clear implementation instructions and examples for future researchers working on the problem of workspace division with the ultimate goal in mind of facilitating the usage of multiple UAV systems.

Specific objectives of this thesis can be outlined as follows:

- Investigate how planning is currently done based on what other researchers describe in their works.
- Analyse the algorithms in the literature that solve the polygon partition problem, classify them, and provide an explanation of their specific use cases, advantages, and drawbacks.
- Implement algorithms for dividing convex and non-convex polygons, analyze and compare produced results using various metrics, and improve the algorithms.
- Use extensive tests of the produced results using randomly-generated polygons to validate the correctness of the algorithms.
- Outline potential future work and notable problems that arose during the time of preparation of the thesis.

I.3 Scope and Limitations of this PhD Thesis

For this thesis, only computer simulations of real-life scenarios were executed. The experiments with real drones were not performed since the simulations are more general and could provide us with more results.

All the simulations are performed in two-dimensional (2D). We consider three-dimensional (3D) geometry to be out of the scope of the thesis.

We also did not consider multipolygons for the simulations in this thesis. The problem of multipolygon workspace decomposition at the moment never arose, and there was only one research work that discussed CPP across multiple disconnected polygons that we could find (Chen *et al.*, 2021). Therefore, we decided to leave this specific case for future work.

It is also worth noting that unrealistic polygons were disregarded. More specifically, we consider the polygons with a too-low ratio of area to perimeter unrealistic and not useful for any practical usage. Nevertheless, the presented algorithms are able to work with these types of geometries but we never considered them in our analysis. We also did not consider polygons with more than 100 vertices and more than 10 holes. Similarly, we omitted the cases with a large number of UAVs or, in other words, area requirements, and the cases when the relative difference between the area requirements was too high. Nonetheless, the presented algorithms are capable of solving such cases but it was necessary to choose a reasonable upper limit for these numbers.

Partition by arcs was also not considered. Only straight segments are used in this thesis to divide polygons. In fact, the usage of arcs for workspace decomposition was never performed to the best of our knowledge. There exist works, however, in the area of computational geometry with the aim of performing polygon partitioning with arcs.

Another important note is that in this thesis we do not develop CPP algorithms and consider them to be out of scope. Nonetheless, we use some metrics related to CPP algorithms to justify using compactness as an appropriate metric for area partitioning.

All the implementations of the presented algorithms were tested using large numbers of randomly generated polygons. There is an issue, though, that the term random polygon is not well defined. There are various algorithms to generate them but due to their nature, they produce bias to some types of polygons, or simply cannot generate some types. For example, some research works reported that their algorithms cannot generate star-shaped polygons (Sadhu *et al.*, 2013) or polygons with complex holes (Hada, 2014).

Finally, the task of designing an algorithm that could find the best possible partition in terms of a chosen metric is challenging. This is why in this thesis we propose solutions that produce solutions *close to optimal*.

I.4 Outline of this PhD Thesis

The present thesis is organized into seven chapters. The following chapters are summarized as follows:

- **Chapter II** presents the state of the art of different types of algorithms for workspace division. We review those research works that are regularly referenced as solutions for the workspace division problem. We also provide information on more recent research works discussing specifically the polygon decomposition. All the algorithms are classified into different groups. In the end, we discuss the drawbacks that motivate the design of new algorithms.

- **Chapter III** provides details about the methodology of implementing algorithms. The required functionality is covered as well as various libraries are compared. Brief information was also provided about how the algorithms were tested. Finally, we describe how trajectories were generated that were necessary to measure several metrics of the quality of obtained partitions.

- **Chapter IV** discusses two algorithms. One is presented by Hert & Lumelsky (1998) for convex polygon decomposition and another one is a novel algorithm based on an analytical

solution. Both algorithms are explained in detail and their implementation details are discussed.

- **Chapter V** discusses two algorithms. The first one is an algorithm from Hert & Lumelsky (1998) for the decomposition of non-convex polygons. Another algorithm is a novel algorithm for dividing non-convex polygons into multiple parts based on a bottom-up approach. Likewise, a detailed description of the algorithms is presented and the results are shown and compared with the other methods.
- **Chapter VI** presents conclusions of this thesis. Achievements of the work performed during writing the thesis are discussed. Potential future work is also described in detail.

II

Previous Work

Algorithms for finding area partition between multiple unmanned aerial vehicles (UAVs) are important in many applications, such as search and rescue missions, surveillance, agriculture, and others. These algorithms typically involve dividing a given area of interest into smaller regions, which can be assigned to individual UAVs for efficient coverage. These algorithms can be divided into several classes.

In Section II.1 we discuss several works about decomposition using Voronoi diagrams. In Section II.2 the most common way found in the literature to split an area is shown — grid partitioning. In Section II.3 we discuss algorithms using region-adjacency graph partitioning. Finally, in Section II.4, algorithms for decomposing non-convex polygons into convex parts are covered as they are frequently used as part of the workspace decomposition algorithms.

II.1 Voronoi diagram

Several works like (Kim & Son, 2020; Sun *et al.*, 2018) use the Voronoi diagram to divide a polygon into several pieces. The idea is as follows. First, a set of lattice points is constructed inside the given polygon and several clusters are found based on the number of parts the polygon has to be split into. A centroid point is found for each cluster. The resulting set of centroids is then used to construct the Voronoi diagrams where the inner points of each resulting area are closer to the corresponding centroid than to any other one. The complete system flowchart from Kim & Son (2020) is shown in Fig. II-1.

The drawback of this algorithm is that it is not possible to specify the areas of the resulting parts. In (Kim *et al.*, 2020), authors proposed a weighted Voronoi diagram for a partition that takes

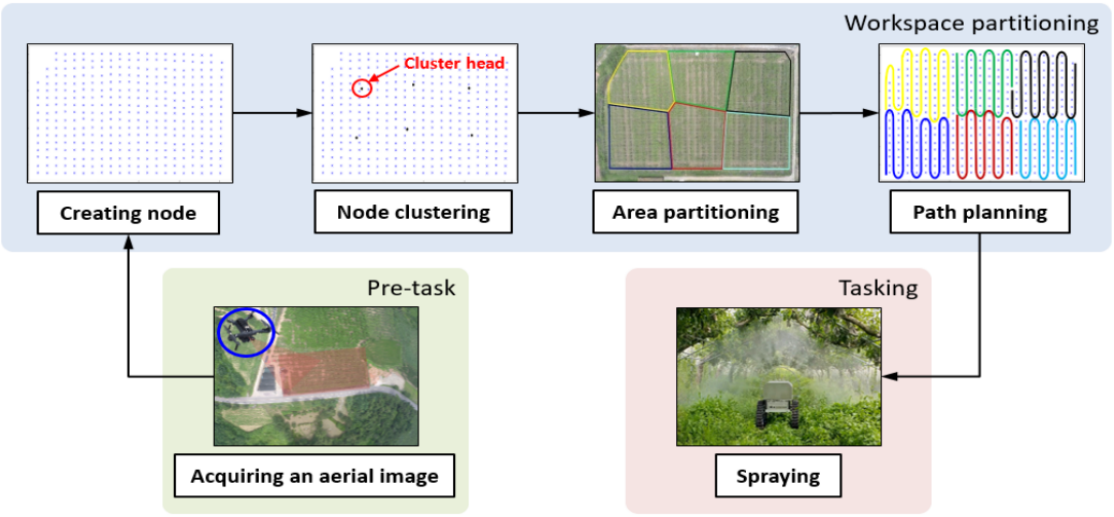


Figure II-1: *System flowchart from Kim & Son (2020).*

into account the different capabilities of the UAVs. A weighted Voronoi diagram differs from a regular Voronoi diagram in that the former considers both the spatial proximity and the weights of the input points when partitioning the space. The weights associated with each input point, in general, represent their importance or influence in the partitioning of the space. And in this case, the weights were used to represent the capabilities of the UAVs. An example is shown in Fig. II-2. The area proportions of the resulting parts will be closer to the given capabilities proportions but it will be still impossible to specify precisely the areas of resulting parts. Moreover, these algorithms were tested only on convex and close-to-convex polygons. There was no information provided on how they will perform on more complex polygons.

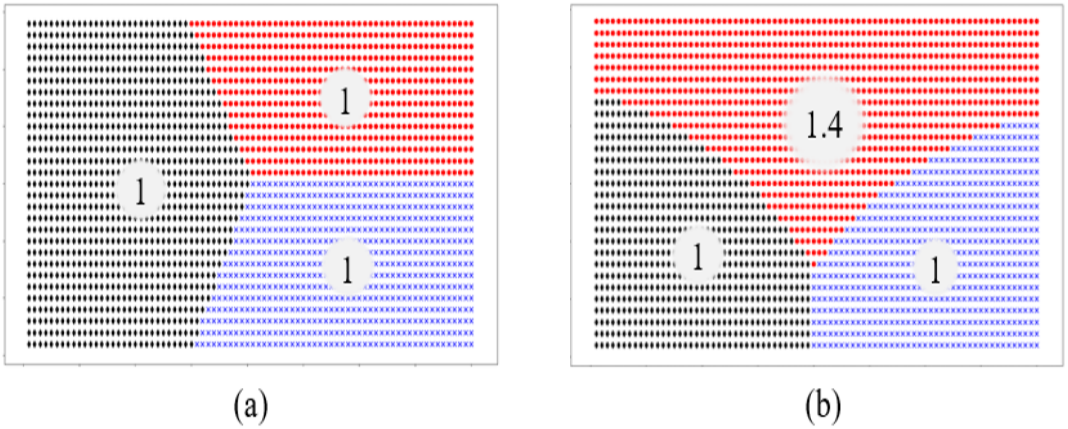


Figure II-2: *The area partition between three robots based on a weighted Voronoi diagram: (a) the area divided into sub-areas by the same weights; (b) the area divided into sub-areas by the weight of 1.4 for one sub-area. Numbers denote weights assigned to each area.*

II.2 Grid partitioning

The most commonly used way to perform area decomposition found in literature is grid partitioning. This method consists in dividing an input polygon into multiple parts to form a grid. Cells of this grid are then divided into several groups, one group per UAV. Usually, researchers con-

sider rectangular areas of interest (Kapoutsis *et al.*, 2017; Barrientos *et al.*, 2011; Gao & Xin, 2019; Dong *et al.*, 2020; Ann *et al.*, 2015; Xu *et al.*, 2019). Most of the works, if not all, that perform grid partitioning, build trajectories over the grids. While the coverage path planning (CPP) for a single UAV is a more common task, there exist various research works on CPP for multiple UAVs as well. An example of such partition with trajectories applied on top of the resulting parts is shown in Fig. II-3. There, each color represents an area assigned to a UAV except the red color which shows the no-fly zones.

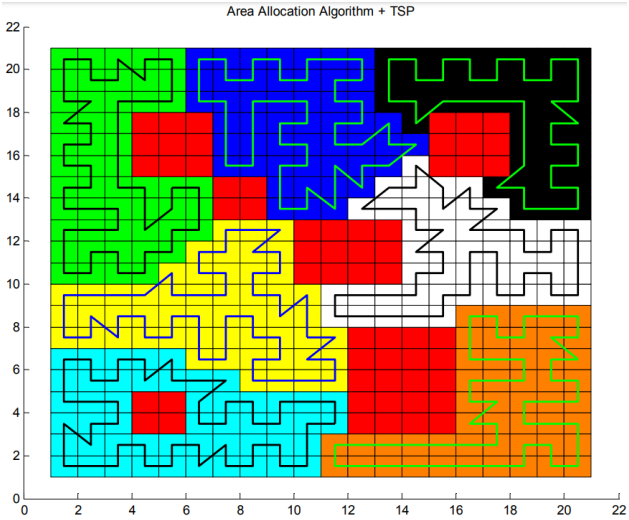


Figure II-3: *An example area partition from Ann et al. (2015) with applied trajectories.*

As an example, Barrientos *et al.* (2011) discusses the use of a multiple unmanned aerial system (UAS) that takes 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 (Newman & Sproull, 1979) to obtain the subareas. This approach has several drawbacks, though. For example, this approach will result in partial cells at the border of the area that will still have to be visited. 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.

In some works, the authors step away from using conventional rectangular cells. For example, Tang *et al.* (2020) used sensor footprints to cover the area. The set of footprints was divided into several clusters of equal sizes, one cluster for one UAV, using an improved k-means clustering algorithm. The procedure is shown in Fig. II-4. This algorithm, though has a drawback, in that the initial partitioning is done by a set of straight lines which will not necessarily give a valid partition in terms of connectivity.

The research performed in Kapoutsis *et al.* (2017) focuses on solving the problem of path planning for a group of mobile robots to cover an area with pre-defined obstacles. The authors introduce a new algorithm that finds the best solution, or at least a close approximation of it, by breaking down the original problem into smaller single-robot problems. The core of the approach is a **D**ivide **A**reas based on **R**obot's initial **P**ositions algorithm (DARP) which divides the grid into equal-area parts based on distances to initial locations of the robots. Each area is then covered by a minimum-spanning tree which ensures complete coverage without any backtracking. This approach helps to reduce the computational complexity of the original problem, making it more practical for real-world applications. The authors conducted an extensive numerical analysis to

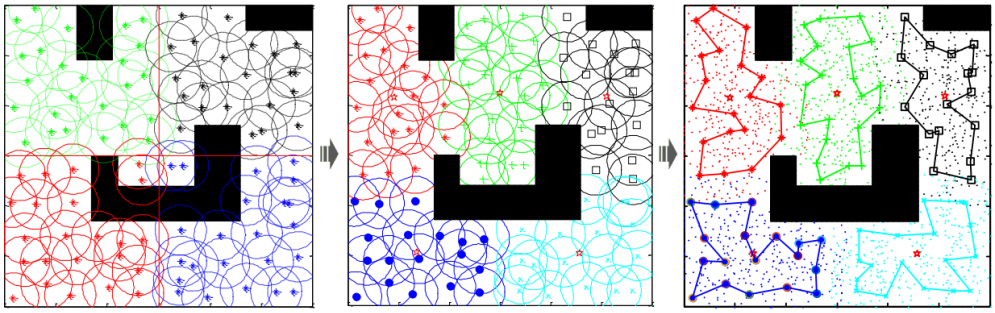


Figure II-4: *Illustration of the algorithm from Tang et al. (2020).*

evaluate the performance of the proposed algorithm, and their results indicate that polynomial curves bound the algorithm's computational complexity for practical input sizes. Overall, the proposed algorithm offers a practical solution to the problem of path planning for mobile robots in complex environments.

This work was later used in Apostolidis *et al.* (2022) discussing a cooperative multi-UAV coverage mission planning platform for remote sensing applications. There the algorithm was extended to take into account different requirements for areas for each UAV. An example of partitioning is shown in Fig. II-5. The only drawback of the approach could probably be the use of a rectangular grid that requires careful consideration of areas' borders as the representation of a complex polygon using a grid can result in discontinuities. Nevertheless, since this is the only open-source algorithm for polygon partitioning, we will use it in Chapter V when comparing the results produced by our algorithms.

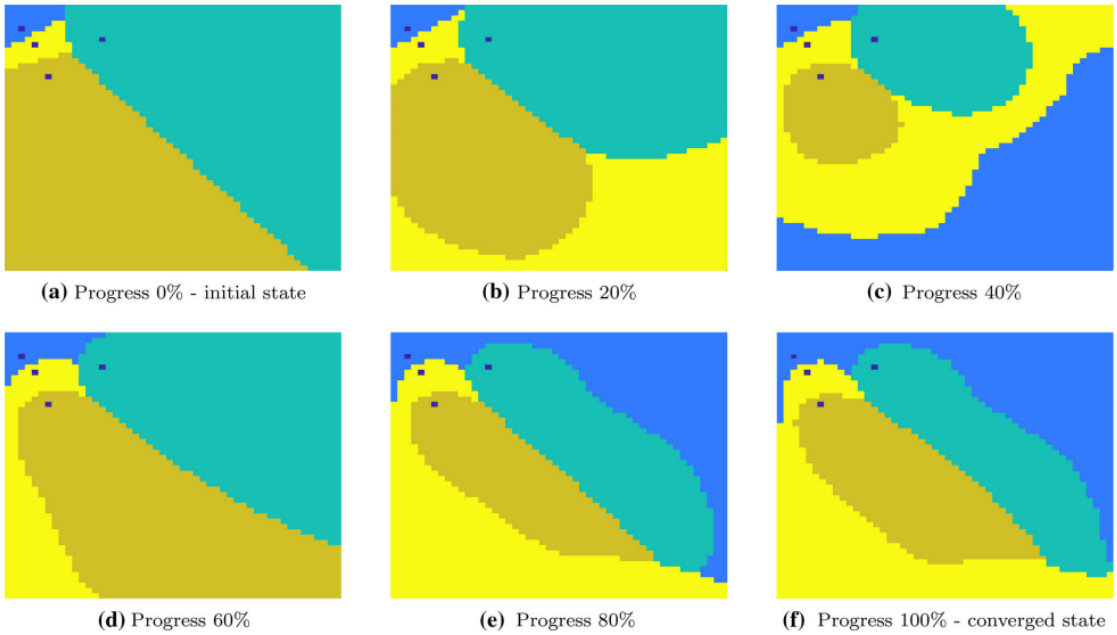


Figure II-5: *Partitioning progress during the course of iteration as seen in Apostolidis et al. (2022). The area requirements are chosen to be equal.*

In Balampanis *et al.* (2017) the authors continue their work started in their previous papers. Here, the area is proposed to be split into a mesh of triangles instead of a rectangular grid. These triangles get constructed from the constrained Delaunay triangulation using Steiner points where Steiner points are points located in a grid-like manner inside the polygon. The obtained grid is then split by using wavefront propagation from the initial positions of UAVs. An adjustment

algorithm for deadlock scenarios is used then.

Not really grid-based, but, probably the most fitting into this class, the algorithm proposed by Pintado & Santos (2020). The idea of the algorithm is to split an area into a set of parallel stripes. These stripes are then grouped together according to the given area requirements. For convex polygons, this approach would yield sound results according to the performed analysis in this paper. But, clearly, this approach has the drawback of producing disconnected areas in many cases when the division is performed for complex polygons. While for some non-convex polygons it is possible to achieve partition without discontinuous areas, this is not always the case. An example of a polygon divided into eight parts with resulting discontinuities is shown in Fig. II-6.

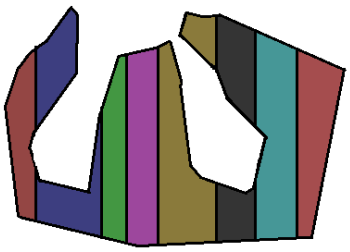


Figure II-6: *Partition of a polygon into eight parts by using the algorithm from Pintado & Santos (2020).*

Wzorek *et al.* (2021) addressed the problem of partitioning a polygon into connected, disjoint sub-polygons, each with a specific area size constraint, for efficient terrain coverage applications in robotics. The proposed formulation included a compactness metric for the generated sub-polygons, which affected the optimality of any generated motion plans, thus increasing the efficiency of robotic tasks. The paper proposed a new algorithm, the *AreaDecompose* algorithm, based on grid cell decomposition and a potential field model, which employed various optimization techniques and post-processing methods. The algorithm's performance was evaluated on a set of randomly generated polygons and compared with a state-of-the-art algorithm (Hert & Lumelsky, 1998) which we will cover in the following section, showing that the proposed algorithm can efficiently divide polygons into regions more compact than the algorithm from Hert & Lumelsky (1998). An example of the area decomposition is shown in Fig. II-7.

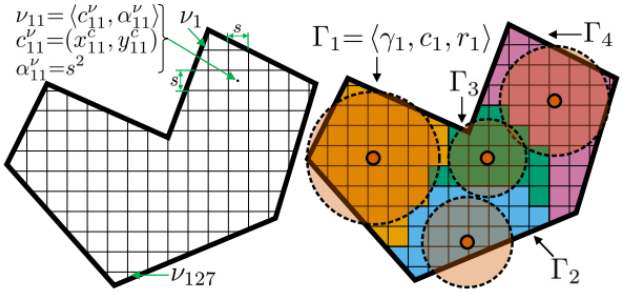


Figure II-7: *Example of area decomposition using grid decomposition and potential fields from Wzorek et al. (2021).*

II.3 Sweep line-based algorithms

Hert & Lumelsky (1998) proposed two algorithms, an algorithm to divide convex polygons and an algorithm to divide non-convex polygons. The algorithm for dividing a convex polygon does it

in a recursive manner. A line with a fixed endpoint is swept until satisfying the area requirements. The resulting two parts are then divided using the same procedure. The process repeats until the polygon is completely divided into parts with areas corresponding to the input area requirements. An example of the resulting partition can be seen at the top of Fig. II-8.

Xing *et al.* (2020) improved the algorithm by post-processing each obtained part. The lines that divided the obtained sub-polygon from the rest of the polygon were moved until the two angles introduced by each line were as close as possible to 90 degrees. The algorithm was tested only on a single rectangle, so more tests are needed to understand if it works for more complex cases. Fig. II-8 shows the test case analyzed in the paper.

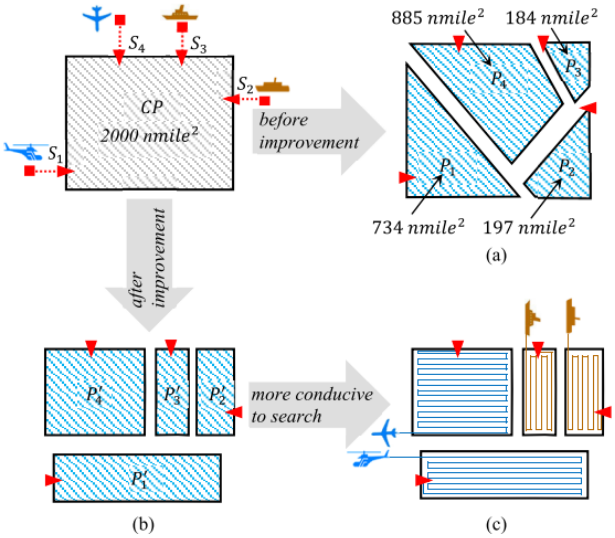


Figure II-8: *Test case of the algorithm from Xing et al. (2020).*

The algorithm for dividing non-convex polygons proposed in Hert & Lumelsky (1998) is based on their algorithm for dividing convex polygons. This algorithm represents a convex polygon partition as a directed region-adjacency graph with nodes as its convex parts. The nodes of the graph are then processed recursively and reorganized into parts according to the given area requirements. When an area of a node is greater than the required area, a sweep-line algorithm is used to get the necessary area. An example of obtaining the partition using the algorithm is shown in Fig. II-9. It's worth noting that the authors do not share how they produced convex decomposition for non-convex polygons for their analysis.

These two algorithms by Hert & Lumelsky (1998) are, probably, the most popular choice in the literature when dealing with workspace decomposition. However, it was noted in several research works that the resulting parts can be of shapes that make them a poor choice for being covered by UAVs.

Berger *et al.* (2016) considered the problem of scanning an area with a team of UAVs. It reuses the algorithm from Hert & Lumelsky (1998) but adds optimization for weights and initial positions on top. The main contribution is 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 where point density represents the level of details of the point cloud generated by the scanning mission. In this work, the algorithm runs multiple iterations to optimize for the specified requirement. An example is shown in Fig. II-10. While the algorithm performs well for the given task, in our work we consider the weights, or area requirements, unlike in the given paper and, hence, we cannot use it.

Hert & Richards (2002) provide a similar algorithm to the one discussed in Hert & Lumelsky (1998), but in this case, a slightly different task is solved. A polygon is split between n mobile

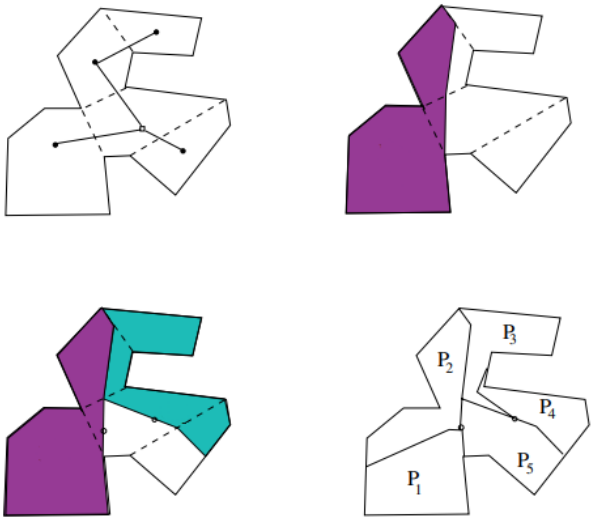
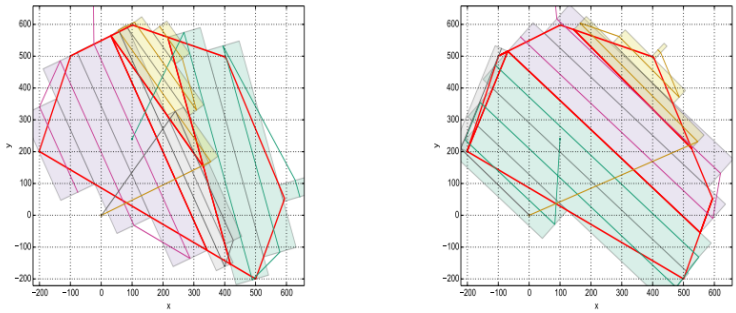
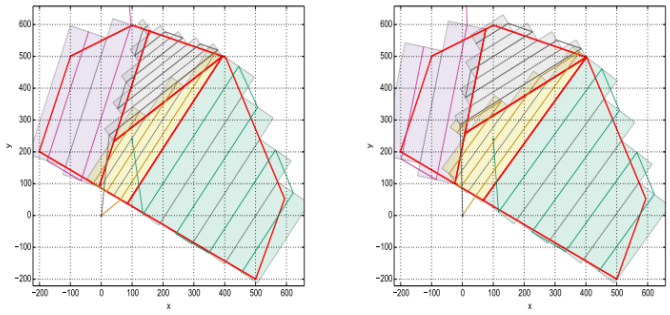


Figure II-9: *An example partition from Hert & Lumelsky (1998).*



(a) After 5 iterations **(b)** After 10 iterations



(c) After 100 iterations **(d)** After 1000 iterations

Figure II-10: *Results of optimization from Berger et al. (2016).*

robots into $n + 1$ parts where the last part should be connected to all the other n parts that are assigned between the robots.

The paper by Hert & Lumelsky (1998) is probably the most popular choice in the literature for the problem of workspace decomposition. We chose to implement both algorithms from that paper and improve them where possible. In the next chapters, the algorithms will be discussed in more detail. The results produced by them together with the comparison against alternative algorithms will be shown in the following chapters.

II.4 Decomposition non-convex polygons into convex parts

Decomposition of non-convex polygons into convex ones is often required by sweep-line-based algorithms. It is also the first step of the algorithm in Hert & Lumelsky (1998). The authors refer to several research works with the algorithms for this task (Chazelle, 1980; Greene, 1983; Hertel & Mehlhorn, 1983; Keil, 1985; Levcopoulos & Lingas, 1984; Tor & Middleditch, 1984).

1. Chazelle (1980) proposed an algorithm that finds a minimum partition of a simple polygon in $O(n^3)$ time where n is the number of vertices. The algorithm, to the best of our knowledge, was never implemented probably due to its complex description. It was also designed for partitioning polygons that do not contain holes.

2. The algorithm by Greene (1983) finds an optimal partition of a simple polygon in $O(n^4)$ time and requires $O(n^3)$ space in the worst case. This algorithm has an implementation in CGAL (The CGAL Project, 2022), but it does not support polygons containing holes.

3. The Hertel & Mehlhorn (1983) algorithm produces the results which are never worse than $2r + 1$ pieces, where r is the number of reflex vertices. And the results are never worse than four times the minimum. The algorithm has various implementations[1,2]. In CGAL[3], for example, it does not work with polygons that contain holes.

4. Keil (1985) presents a dynamic programming algorithm for finding the partition of a simple polygon into convex parts in $O(N^2 n \log n)$ where n is the number of vertices and N is the number of reflex angles. Similar to other algorithms, it does not work with polygons containing holes.

5. Levcopoulos & Lingas (1984) provides a heuristic for partitioning a polygon into convex parts with a generalization to include polygons with polygonal holes. Both variants give minimum line lengths and run in $O(n \log n)$ where n is the number of vertices.

6. Tor & Middleditch (1984) proposed an algorithm that can decompose concave polygons into convex parts in linear time if the difference between the convex hull and the polygon itself is a convex area itself. The algorithm is a difference-decomposition method meaning that it returns the difference between the non-convex polygons and their corresponding convex hulls which is not what we are looking for. In the worst case, the algorithm has quadratic complexity.

7. Finally, the paper Lien & Amato (2006) proposes an algorithm for an approximate convex decomposition of polygons. And since the algorithm described in this chapter requires the parts of the polygon to be strictly convex, this particular algorithm does not suit our needs.

Delaunay triangulation can also provide division of non-convex polygons into convex parts but the number of resulting parts is far from optimal.

[1] https://github.com/ivanfratric/polypartition
[2] https://github.com/azrafe7/hxGeomAlgo
[3] https://doc.cgal.org/latest/Partition_2/group__PkgPartition2Ref.html

III

Implementation and Evaluation

During the course of writing the thesis, we implemented several algorithms in the area of computational geometry. These algorithms were described in pseudocode in their respective papers. On multiple occasions, the descriptions of these algorithms were too general, and details were missing that could allow us easily to implement them. This is why we think that it is important not only to show a general description of an algorithm but also to provide a sufficient amount of details on how to implement it. In this chapter, we take a look at the functionality that will be required when implementing the algorithms in Section III.1. In Section III.2 a comparison of different libraries will be shown. We discuss which libraries satisfy various requirements. In Section III.3, we describe how the algorithms were tested. In Section III.4, it is explained how trajectories were assigned to the resulting partitions in order to obtain various metrics. And in Section III.5, various metrics are discussed that were used for validating the algorithms.

III.1 Required functionality

III.1.1 Shapes

The algorithms discussed in this thesis require the use of the same set of geometrical objects. The central object is, of course, the polygon. We are interested in polygons that could contain any number of holes since it is easy to imagine a real-life case where the area to be partitioned contains obstacles or no-fly zones (NFZs). We will need the polygons to carry information about the area they cover as well as the perimeter of the outer boundary of a polygon. The perimeter will be used for checking the quality of obtained results.

Apart from the polygons, we will also need access to segments defining their boundaries. If

a library does not provide the corresponding data type, it could, for example, give access to an ordered list of points defining the border. The segments could be then reconstructed from these points. And, finally, the points themselves should be a part of the library. In this thesis we consider only planar geometry, hence if a library works with three or more dimensions, we could simply omit the unnecessary ones.

III.1.2 Operations

Apart from the basic set of geometric objects, the library of interest should have implementations of set-theoretic operations on these objects. These operations are summarized in the following list:

- polygon area calculation
- point-in-polygon — checks if a point is within a polygon
- segments intersection — while easy to implement, it is expected that a geometry-related library has functionality that performs this check
- polygonal union — for adding polygons into a larger one
- polygonal difference — for subtracting parts of a polygon
- polygonal intersection — for obtaining a common area of two or more polygons
- calculating orientation — for a list of points we will need to check if they are oriented clockwise, counterclockwise, or if they are collinear

III.1.3 Precise calculations

An important requirement for a library in the area of computational geometry is precise calculations. That means that the coordinates of the geometric objects cannot be rounded or represented imprecisely with floating point numbers. As it will be shown later, the majority of the libraries do not support precise calculations.

A simple example to demonstrate a problem caused by precision would be as follows. The Shapely library is used for the demonstration. Let us take a point defined as 'Point(2/3, 2)' and a triangular polygon defined as 'Polygon([(0, 0), (1, 0), (1, 3)])' as depicted in Fig. III-1. It is clear that the point should lie on one of the segments of the polygon. We can check the distance between the polygon and the point as 'Point(2/3, 2).distance(Polygon([(0, 0), (1, 0), (1, 3)]))' and the result we get is satisfactory, '0.0'. But, when checking if the point intersects the polygon by 'Point(2/3, 2).intersects(Polygon([(0, 0), (1, 0), (1, 3)]))', the result will be incorrect, 'False'. These types of errors tend to propagate through the run of the algorithms and can make it very hard to debug them. If the library were to work with precise data types such as 'Fraction', this problem would not occur. Hence, it is very important that the calculations be precise.

For more information on what can go wrong and why with the algorithms in the area of computational geometry due to using floating-point arithmetic, we refer an interested reader to the works like (Schirra, 1998) and (Kettner *et al.*, 2008). In the latter, the authors provide examples of two algorithms, one for calculating Delaunay triangulation, and the other one for computing the convex hull. It is shown how the algorithms can fail with different examples and how these examples can be constructed.

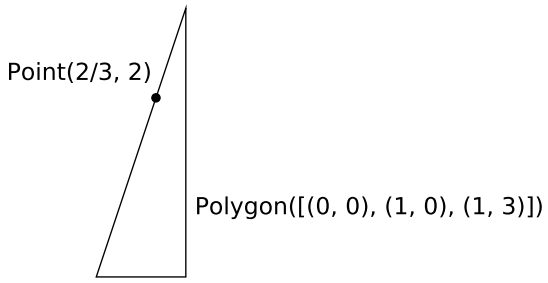


Figure III-1: *In Shapely, due to precision errors, when checking if a point with coordinates* $(2/3, 2)$ *lies on a segment of a polygon defined by the coordinates* $[(0, 0), (1, 0), (1, 3)]$*, the result unexpectedly will be* *False*.

III.1.4 Graph-related computations and hashability

In the algorithms presented in this thesis, the input polygons are split into multiple parts, such as convex parts or triangles from Delaunay triangulation. It is necessary that the neighbors of each part could be easily retrieved as well as the edges between these parts. In order to do so, we require data structures that act either like graphs or dictionaries. And in order to use geometric objects as keys of these dictionaries or as nodes in the graphs, they should be hashable. It turns out, as it will be shown later, there exist libraries that for various reasons use mutable geometric objects which makes them unhashable and, therefore, unusable for our purposes. And as for graph-related computations, it will be shown later that there are no libraries in the area of computational geometry that provide functionality for planar region-adjacency graphs. For this reason, the libraries that could be used in order to complement the missing functionality will be looked at separately.

III.1.5 Tests

Even the most popular geometry-related libraries are not free of issues. Very simple input data could result in incorrect results or simply crash the program. In order to avoid this, authors of these libraries write tests to check the correctness of their code. But in the majority of cases, these tests cover just a tiny fraction of possible input parameters that could be passed to the functions of those libraries. When a bug is found, the authors simply add another test case, and the process repeats. Alternatively, property-based testing can provide a better way to test the functionality of the libraries. By generating multiple randomly-generated input parameters, and checking the basic properties of the results from performed operations, one could find more hidden issues with the code. We think that a good geometry-related library should be covered by tests, and ideally, these tests should be property-based which, unfortunately, is a very rare occurrence.

All of the important functionalities required from the library are shown in Table III-1 together with their brief descriptions.

III.2 Libraries comparison

One of the first challenges when implementing an algorithm is choosing a library that satisfies the necessary requirements. There exist not many geometry processing libraries, so we can cover all of them here. It is worth noting that we do not include libraries whose main goal is plotting such as Matplotlib as they include a very limited number of tools to work with geometries.

Table III-1: *Description of required functionalities*

Functionality	Description
Point	basic building block for geometric objects
Segment	used for defining polygons' boundaries and polylines dividing the polygons
Polygon	representation of workspace
holes	representation of no-fly zones and obstacles
point-in-polygon	common geometric query returning a boolean
segment intersection	common geometric query returning a point
precise calculations	ensures correctness
hashability	allows unique identification by hash values which allows fast data access
tests	helps to ensure the reliability of the library
property-based tests	helps to identify edge cases that may not be covered by traditional unit tests

III.2.1 Shapely

Shapely[1] is probably the most popular library used for set-theoretic analysis and operations on planar geometry objects. The library uses heavily the GEOS library (GEOS contributors, 2021) under the hood which is itself a port of the Java Topology Suite (JTS). It does provide all the necessary geometry objects such as *Point*, *Segment*, *Polygon*, and others that will be required for our use case. It also provides multiple set-theoretic functionality and various functions that we could benefit from such as Delaunay triangulation. Unfortunately, apart from all the positive things this library has to offer, it has multiple drawbacks. First of all, the geometries as of the moment of writing the thesis are mutable and, therefore, unhashable. This makes it impossible to use them as nodes in the region-adjacency graphs, as keys in dictionaries, or as items in sets. This, though, should change as soon as Shapely 2.0 is published [2]. Another drawback is the lack of support for precise calculations. This and the lack of properly tested code causes multiple bugs to emerge when testing the algorithms on multiple random input geometries. For example, when using the Delaunay triangulation, we encountered multiple issues with it that did not let us use it without causing too many problems [3].

III.2.2 PyGEOS

PyGEOS[4] is a wrapper around the GEOS library (GEOS contributors, 2021) that uses NumPy to provide performance boost when working with sequences of geometrical objects. The GEOS itself (Geometry Engine - Open Source) is a C++ library for performing geometric operations on spatial data. It is widely used in many popular open-source GIS software packages, such as PostGIS and QGIS. It is worth noting that PyGEOS has been merged with Shapely in December 2021 and will be released as a part of Shapely 2.0.

The library contains functions that create NumPy arrays of basic types such as points, polygons, and segments. But, there is no way to, for example, instantiate a single polygon unless it is wrapped in the array. This makes it not convenient for our use case. There is, however, a way to simply use *Geometry* class, but it takes a well-known text (WKT) or well-known binary (WKB)

[1] https://shapely.readthedocs.io/
[2] Practical steps towards Shapely 2.0 — https://github.com/shapely/shapely-rfc/pull/1
[3] https://github.com/shapely/shapely/issues/764
[4] https://pygeos.readthedocs.io/

representation of a geometry object where WKT is a markup language for representing geometry objects and WKB is its binary equivalent (Herring *et al.*, 2011). The library has all the necessary operations. The objects are hashable but the geometries are considered to be equal if and only if their WKB representations are equal which has a negative effect on comparing identical geometrical objects with different orders of vertices. Finally, the library has tests but not property-based ones.

III.2.3 Geometry3D

The Geometry3D[5] is a computational geographics library with all the computations performed in three-dimensional (3D) space. It contains all the basic geometry classes but the polygons can be only convex and cannot contain holes. Operations such as the intersection of two different geometric objects are included. This library, however, makes an assumption that any two values that are close to each other with some epsilon, the default being 1e-10 is the same value[6]. This is not ideal and will cause problems in many cases. Finally, all the objects can be hashed, and the library is tested but it does not use property-based tests.

III.2.4 pysal

PySAL[7] is a spatial analysis library created for geospatial data science applications. It contains several modules, one of which is *libpysal.cg*, a module with computational geometry functionality. This module contains all the basic geometry types and the *Polygon* object can contain holes. The *Polygon* also has a method *contains_point*. There exists a set of functions to get intersections between two geometries, such as *libpysal.cg.get_segments_intersect*. The library does not support precise calculations. In fact, all the coordinates passed to the objects' constructors get immediately converted to floating point values. The hashability is not supported for all the geometry objects. Only a *Point* object is hashable. And, finally, the library does not have any tests implemented.

III.2.5 geometer

The *geometer* library[8] is used for projective geometry — a study of properties of geometric objects that are invariant with respect to projective transformations. The library supports all basic geometry data types such as *Point*, *Segment*, and *Polygon*. Apart from two-dimensional (2D) shapes it also provides support for various shapes in 3D space. The *Polygon* class does not support holes which are required for our use case. The library supports such operations as point-in-polygon check — 'Polygon.contains(Point)', the intersection checks — 'Segment.intersect(Segment)', 'Segment.intersect(Polygon)'. At the moment of writing the thesis, precise calculations are not implemented. Though, there was an interest from the contributors of the library to add support of *Fraction* data types [9]. The data types of this library are unhashable and, therefore, cannot be used inside sets or as keys of dictionaries which makes it troublesome to use them in graph-related computations. Finally, the library is covered by tests but no property-based ones.

[5] https://github.com/GouMinghao/Geometry3D/
[6] https://geometry3d.readthedocs.io/en/latest/example_float.html
[7] https://pysal.org/libpysal/api.html
[8] https://geometer.readthedocs.io/en/stable/
[9] https://github.com/jan-mue/geometer/issues/18

III.2.6 CGAL

CGAL[10] (The CGAL Project, 2022) is a C++ library of a variety of computational geometry algorithms. It also has Python bindings [11] that cover some parts of its original functionality. The library contains all the basic geometry object classes including polygons with holes represented by *Polygon_with_holes_2* class. It also contains all kinds of operations including the ones such as *bounded_side_2* that checks point inclusion in a polygon and *intersection* to get an intersection of any pair of geometry objects. CGAL does support precise calculations. It has various number types that can be evaluated lazily. And, finally, the library is extensively tested, but without the usage of property-based tests.

III.2.7 scikit-geometry

Scikit-geometry[12] is a library for geometry-related computations that takes most of its functionality from CGAL The CGAL Project (2022). It implements the basic geometry types in both two and three dimensions such as *Point2*, *Segment2*, *Polygon*, and specifically for polygons with holes — *PolygonWithHoles*. All the geometry objects have basic operations implemented. The precise calculations, however, are not supported unlike in the CGAL itself. The objects are hashable. And the library is partially covered by tests, though not property-based.

III.2.8 SymPy

The SymPy[13] library's primary goal is to allow a symbolic mathematical computation. Among others, it includes a geometry module that provides such objects as *Point*, *Line*, *Segment*, *Ray*, *Ellipse*, *Circle*, *Polygon*, *RegularPolygon*, and *Triangle*. Unfortunately, the classes that define polygons do not support holes. The library has the functionality for checking if a point is inside a polygon provided by the *encloses_point* method of the polygon classes. To calculate the intersection between any geometry objects one can use the *intersection* function from the *util* submodule.

Unlike many other libraries, this library does support precise calculations when using the *Rational* class provided by SymPy itself. The objects are also hashable which makes it easy to use them in dictionaries and graphs. And the library itself is covered by tests that are not property-based, though. And since the library is very complex, the number of reported issues as of the time of writing the thesis is almost four thousand.

III.2.9 gon

Gon[14] is a relatively new and unknown library for processing planar geometry objects. It originated from the dissatisfaction caused by multiple disadvantages of other libraries. Unlike its popular counterpart, Shapely, the geometries are immutable and hashable which allows us to use them as, for example, dictionary keys or nodes in a graph. The library includes all the necessary geometry data types as well as operations on them. Access to segments of the border of the polygon is available, as well, though the library does not have any way of specifying which segment to return first. The correctness of performed operations is ensured by property-based tests covering it. And, finally, this library supports precise calculations. To enable them one would only have to use integer coordinates or coordinates of *Fraction* data type.

[10] https://www.cgal.org/
[11] https://github.com/cgal/cgal-swig-bindings
[12] https://scikit-geometry.github.io/scikit-geometry/
[13] https://docs.sympy.org/
[14] https://gon.readthedocs.io/

III.2.10 Other libraries

There is also a number of other well-known geometry-related libraries that have very limited functionality. For example, the *scipy-spatial*[15] submodule of *SciPy* library has a number of data structures such as KDTrees, and spatial algorithms such as Delaunay tessellation in N dimensions or Voronoi diagrams built on the surface of a sphere.

Polliwog[16] is both 2D and 3D computational geometry library. It includes vectorized geometric operations implemented in pure NumPy which makes it fast when dealing with operations on multiple geometry objects at once. It provides only such objects as polygonal chains, planes, lines, and boxes, which makes it unsuitable for our use case.

III.2.11 Comparison

The table with the comparison of all the aforementioned libraries and their features can be seen in Table III-2.

Table III-2: *Feature comparison of various libraries working with geometry. The library that was selected as the best fitting our purposes is highlighted.*

Library	Point	Segment	Polygon[17]	holes	point-in-polygon	segments intersection	precise calculations	hashability	tests	property-based tests
Geometry3D	Yes	Yes	No	No	Yes	Yes	No	Yes	Yes	No
geometer	Yes	Yes	Yes	No	Yes	Yes	No	No	Yes	No
PySAL	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No	No
Shapely	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes	No
SymPy	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	No
scikit-geometry	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	No
PyGEOS	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	No
CGAL	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
gon	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Given the results summarized in that table, it was decided to use the *gon* library as it fits best for our purposes. It provides all the necessary data classes, operations on them, supports precise calculation, and is thoroughly tested.

III.2.12 RAG functionality

As it will be shown later, we will need not only information about the coordinates and sizes of the parts of the polygon but also their relative position in it. For each part, we will need to know what its neighbor parts are and what common edge they have. Unfortunately, there are practically no libraries that combine the functionality of computational geometry objects and algorithms with that of graph theory. While the CGAL library has graph-related calculations, it does not include region-adjacency graph (RAG) functionality. For this reason, NetworkX library (Hagberg *et al.*, 2008) was used for the implementation of a RAG. This library allows all the basic graph-related tools which allowed for an easy process of extension of its functionality.

[15] https://docs.scipy.org/doc/scipy/reference/spatial.html
[16] https://polliwog.dev/

III.3 Tests

The lifecycle of successful software products starts from the top-level system specification and continues with the derivation of its subsequent requirements. Every requirement must be traceable and tested. Testing must examine code correctness for valid inputs, but also in the absence of valid data or regions of discontinuity. For this, along with the software design, a test plan must be developed (Spitzer & Spitzer, 2000).

The benefits of rigorous software development and testing include both functional and non-functional software performance assurance. Effective software testing can ensure the software quality as well as make the developer work more efficiently in their future developments (Wang, 2004).

In aeronautics and critical infrastructure software, testing is especially important. For this reason, companies developing critical and certified software shall follow one of the existing standards to ensure software quality and compliance. In aeronautics, the RTCA DO-178 defines the software considerations in airborne systems and equipment certification. In other areas, standards like the ISO/IEC 9126 are known as one of the most robust software quality standards.

The area partition software proposed in this work is not safety-critical and does not need any certification. Nevertheless, we have followed an exhaustive test methodology to prove, beyond a reasonable doubt, that the code is error-free, reliable, and follows the planned specification and requirements.

III.3.1 Property-based tests

Modern libraries that work with geometry-related computations check the correctness of the produced results using tests with fixed sets of input parameters. An alternative approach is to use property-based tests where the input parameters are generated in a random fashion. This approach helps find the issues with the code that the regular tests with fixed input data cannot.

When implementing the algorithms shown in this thesis, we used the *hypothesis*[18] Python library (MacIver *et al.*, 2019). This library can generate arbitrary data that matches specifications provided by a user. It then takes the generated data and performs operations specified by a user. If *hypothesis* finds an example that does not satisfy the tested condition, it simplifies the example by size and value until a much smaller example is found that still causes the same problem. This process is also called *shrinking*. It allows a developer to understand easier the root cause of a failure which can save time and effort compared to manually trying to create a small test case from scratch.

As an example, let us take a function that calculates the slope and the y-intercept values of a line defined by two points. The function is shown in Listing III-1

A simple test that we could start with would check if the original points actually lie on the line built with the obtained slope and intercept values. An implementation of this test is shown in Listing III-2.

Running this test will result in an error when both the start and the end points are equal to zero. A *ZeroDivisionError* will be thrown on the second line of Listing III-1 when calculating the slope. To take into account the cases when X-coordinates of both points are equal we can update the function as shown in Listing III-3 and the test to ignore from now on these cases as shown in Listing III-4. After this update, the tests will pass.

[18] https://hypothesis.readthedocs.io/

Listing III-1: Function calculating slope and intercept values based on two points.

```
def slope_intercept(start: Point, end: Point
                    ) -> tuple[Fraction, Fraction]:
    slope = (end.y - start.y) / (end.x - start.x)
    intercept = end.y - slope * end.x
    return slope, intercept
```

Listing III-2: Testing if points lie on the line.

```
from hypothesis import given

@given(start=points,
       end=points)
def test_points_on_line(start: Point,
                        end: Point) -> None:
    slope, intercept = slope_intercept(start, end)
    assert start.y == slope * start.x + intercept
    assert end.y == slope * end.x + intercept
```

Listing III-3: Updated function calculating slope and intercept values based on two points.

```
def slope_intercept(start: Point, end: Point
                    ) -> tuple[Fraction, Fraction]:
    if start.x == end.x:
        raise ValueError("The points cannot have equal x-coordinates.")
    slope = (end.y - start.y) / (end.x - start.x)
    intercept = end.y - slope * end.x
    return slope, intercept
```

Listing III-4: Updated test for checking if points lie on the line.

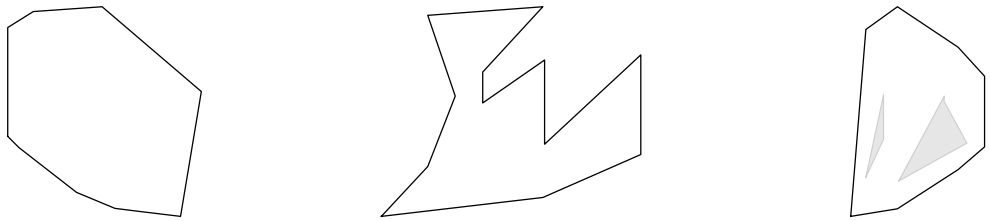
```
from gon.base import Point
from hypothesis import given

@given(start=points,
       end=points)
def test_points_on_line(start: Point,
                        end: Point) -> None:
    assume(start.x != end.x)
    slope, intercept = slope_intercept(start, end)
    assert start.y == slope * start.x + intercept
    assert end.y == slope * end.x + intercept
```

III.3.2 Random polygon generation

Since property-based testing relies on the generation of random data, it is necessary to use an algorithm capable of producing random polygons. Strictly speaking, the phrase "random polygon" does not have a proper definition. And there exist multiple algorithms to generate the random polygons, such as Zhu *et al.* (1996); Auer & Held (1996); Dailey & Whitfield (2008); Sadhu *et al.* (2013); Hada (2014); Gewali & Hada (2015); Pati *et al.* (2015); Nourollah & Movahedinejad (2017); Zhigalova (2017).

In the course of writing this thesis, we used the *hypothesis-geometry* library (Ibrakov, 2022). In this library, the generation of polygons is done by trial and error. Sequences of unique points are generated of a specified data type. These sequences are then filtered according to the specified

restrictions on the number of points in the border and in the holes. When necessary and if possible, extra points are removed to satisfy the requirements. Examples of generated polygons are shown in Fig. III-2.



(a) Convex polygon (b) Non-convex polygon (c) Polygon with holes

Figure III-2: *Examples of randomly-generated polygons.*

III.4 Trajectories assignment

In this thesis, in order to validate the results produced by the algorithms, we apply trajectories over generated sub-polygons. For that, we generate trajectories of a back-and-forth pattern using a ready program from Royo *et al.* (2014). 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 Fig. III-3.

As can be seen, a part of the trajectory lies outside the polygon. This is undesirable since when there will be multiple unmanned aerial vehicles (UAVs) covering adjacent areas, there will be some overlapping which could result in a crash. In order to avoid it, we shrink the polygon by

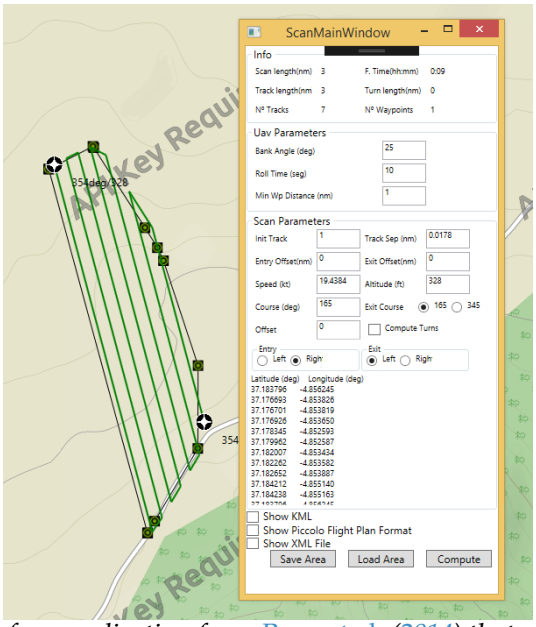


Figure III-3: *Screenshot of an application from Royo et al. (2014) that calculates trajectories for a given polygon.*

a half-width of the track separation before generating the trajectories.

The course direction is chosen to be perpendicular to the direction of the polygon's width as this minimizes the number of turns (Nielsen *et al.*, 2019). The track separation is calculated from the width of the observed area. And the width of the observed area can be calculated from the equation (1) of Maza & Ollero (2007):

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

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.

III.5 Metrics

To evaluate the quality of obtained results we propose using **compactness** – a metric defining how close the area is to a circle. Compactness was proposed as the primary metric in the paper Hert & Lumelsky (1998). There the authors defined it as $Area(P)/Perimeter(P)$. This metric was also proposed in Polsby & Popper (1991) and Schwartzberg (1965) discussing gerrymandering.

In this thesis, we slightly change the metric to make it unitless and define it as follows:

$$C(P) = \frac{\sqrt{Area(P)}}{Perimeter(P)} \tag{III.2}$$

Alternatively, we define normalized compactness so that the best possible compactness for a circle would be equal to one:

$$C_n(P) = \frac{2\sqrt{\pi Area(P)}}{Perimeter(P)} \tag{III.3}$$

Additionally, we propose the following metrics related to trajectories:

- **time of flight** – time that a UAV takes to cover an area following a generated trajectory from start to finish;
- **number of turns** – how many turns a UAV has to make when following the generated trajectory;

These metrics are calculated using the approach presented in Section III.4.

IV

Convex Polygon Decomposition

Convex polygons play a crucial role in workspace decomposition, as they represent simple and obstacle-free geometric spaces. These regions can also emerge during the process of decomposing non-convex areas among multiple UAVs. By developing algorithms specifically tailored for convex areas, which yield superior results, their application can be extended to the decomposition of non-convex spaces. Therefore, it is imperative that we address the challenge of decomposing convex areas separately.

In this chapter, we present two algorithms, the IHLC algorithm which stands for **I**mproved algorithm by **H**ert and **L**umelsky for **c**onvex polygons, and a novel algorithm capable of workspace decomposition when the workspace is defined by a convex polygon. The first algorithm is discussed in Section IV.1. The latter algorithm is based on an analytical solution for the most compact partition of a convex polygon into two parts. This algorithm will be referred to as *PDAN* standing for **P**olygon **D**ecomposition based on **AN**alytical bi-partition. The algorithm is explained in Section IV.2.1. In Section IV.3 results produced by both algorithms are analyzed in terms of quality. The performance of the algorithms is also discussed.

IV.1 Improved Hert and Lumelsky's algorithm for convex polygons

IV.1.1 Theory

The IHLC algorithm is based on a sweep-line algorithm that performs a sequence of splits into two parts. The initial locations of the unmanned aerial vehicles (UAVs) can be taken into account or ignored.

Given a convex polygon $\mathcal{P}$, a set of q points also called *sites* $S_{1..q}$ located in the interior or

on the boundary of the polygon, and a set of values $c_{1..q}$ with $\sum c_i = 1$, the task is to divide $\mathcal{P}$ into q non-overlapping sub-polygons $\mathcal{P}_{1..q}$ in such way that $Area(\mathcal{P}_i) = c_i Area(\mathcal{P})$ and $S_i \in \mathcal{P}_i$ $\forall i \in [1, q]$. A site represents an initial location of a robot or, generally speaking, a point that has to be included in the resulting sub-polygon. The values $c_{1..q}$ represent the area requirements for each part. For simplicity, a function *AreaRequired* is defined as taking a site as input and returning the corresponding area requirement. And $S(CP)$ is defined as a set of sites assigned to a currently analyzed polygon part, CP.

Hert & Lumelsky (1998) propose the partition to be achieved by a sequence of divisions into two parts. The algorithm for dividing a polygon into two parts with several adjustments is shown in Alg. IV.1.

Before providing the first algorithm, we explain the pseudocode nomenclature used in this thesis. Every algorithm description starts with the list of input data denoted by the "Require" keyword. Comments are added after the "▷" symbols. Keywords related to conditional clauses and loops are highlighted using bold text, such as "while", "for", "do", "if", "else if", "then", and "else". Keywords related to returning values are also highlighted using bold text such as "return" for regular functions and "yield" for generator functions that allow for potential iteration over infinite iterables and do not require storing every value in memory. Variables assignment is represented by either $\leftarrow$ or $=$ depending on the context. The function "next" is used to retrieve the next value from the iterator passed to it. Values within square brackets represent an ordered array or a list. Finally, negative numerical indices represent the position of the value in an array when counting from the end. So, for example, V_{-2} is the penultimate element.

The algorithm takes as input parameters a polygon $\mathcal{P}$, an ordered list of vertices and sites $W_{1..m}$ where vertices define the polygon border, and separately a list of sites $S_{1..q}$ ordered according to their appearance in $W_{1..m}$.

First, a segment $L = (L_s, L_e)$ is initialized so that the starting point L_s is equal to W_1 and the endpoint L_e is equal to S_1. Then, while having fixed L_s, the point L_e gets sequentially assigned to all the vertices from W in the same order starting from S_1 until one of three possible conditions is satisfied:

1. Area on the right of the segment equals the total area requirement of all the sites on the right side of the segment — $Area(P_L^r) = AreaRequired(S(P_L^r))$. In this case, the area requirement is satisfied and both parts P_L^r and P_L^l have at least one site since the point L_e never goes beyond the last site on the border S_n.

2. There is only one site on the right side of the segment, and its corresponding requirement is less than the area on the right side — $S(P_L^r) = S_1$ and $Area(P_L^r) > AreaRequired(S(P_L^r))$. In this case, the point L_s is moved counterclockwise until the areas get equal.

3. The area on the right side of the segment contains all the sites, but it is less than their total area requirement — $Area(P_L^r) < AreaRequired(S(P_L^r))$ and $L_e = S_q$. In this case, L_s is moved clockwise until reaching the division with the areas equal to corresponding area requirements.

This algorithm is then can be called recursively for those obtained polygon parts that contain more than one site.

Algorithm IV.1: *The algorithm for dividing a convex polygon into two smaller polygons*

Require: $\mathcal{P}$ — convex polygon,
$W_{1..m}$ — ordered list of the vertices defining the border of $\mathcal{P}$,
$S_{1..q}$ — a list of sites ordered according to their order in the list W

Assume $S_1 = W_k$; $L \leftarrow (W_1, W_k)$
$S(P_L^r) \leftarrow \{S_1\}$ ▷ sites that belong to P_L^r — polygon to the right of L
while $Area(P_L^r) < AreaRequired(S(P_L^r))$ and $L_e \neq S_q$ **do**
  if $k > 1$ and $(W_{k-1} \in S_{1..q})$ **then**
    $S(P_L^r) \leftarrow S(P_L^r) \cup W_{k-1}$
  $k \leftarrow k + 1$
  $L_e \leftarrow W_k$
if $L_e = S_1$ and $Area(P_L^r) > AreaRequired(S(P_L^r))$ **then**
  Move L_s counterclockwise along the border of P until $Area(P_L^r) = AreaRequired(S(P_L^r))$
else if $L_e = S_q$ and $Area(P_L^r) < AreaRequired(S(P_L^r))$ **then**
  Move L_s clockwise along the border of P until $Area(P_L^r) = AreaRequired(S(P_L^r))$
else
  Find point t on (W_{k-1}, W_k) such that if $L_e = t$ then $Area(P_L^r) = AreaReqruired(S(P_L^r))$
  $L_e \leftarrow t$
$P_L^l = P - P_L^r$
$S(P_L^l) = S(P) - S(P_L^r)$
return P_L^l, P_L^r, $S(P_L^l)$, $S(P_L^r)$

IV.1.2 Algorithm

The implementation of the IHLC algorithm follows the pseudocode shown in Alg. IV.1 without any changes. The implementation, however, was extensively covered by tests which allowed us to ensure the correctness of the produced results.

During the process of the implementation, it became apparent that the input data which includes the convex polygon and a list of sites have to be of a precise data type. When using regular floating-point data types, the algorithm either resulted in the incorrect partition or simply crashed. Fraction data type was used instead, and the extensive tests proved that only with this type, we can ensure 100% of correct results.

IV.1.3 Examples

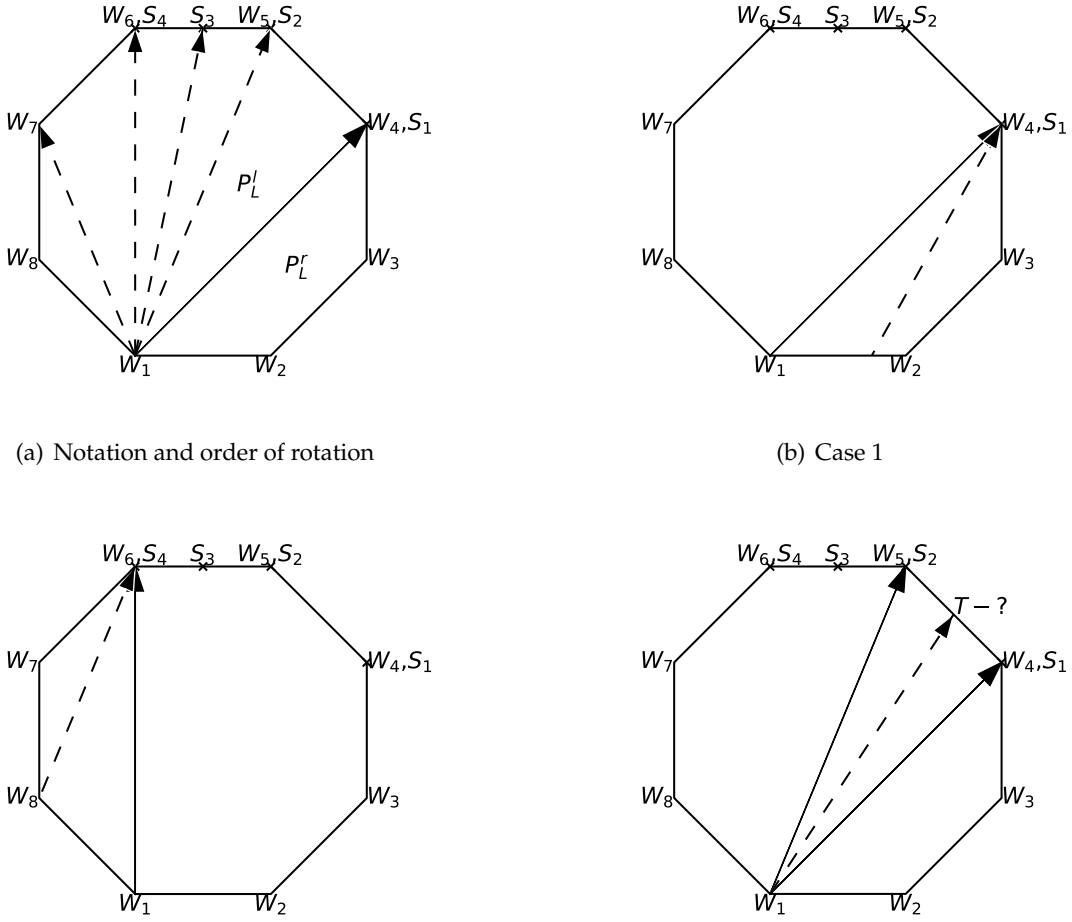
Fig. IV-1 shows an example polygon and various steps and cases from the IHLC algorithm. Fig. IV-1(a) shows a convex polygon defined by eight vertices, $W_{1..8}$. There are four sites, $S_{1..4}$, located on the border and ordered according to the order of vertices W. The line-splitter is initialized on the vertices W_1 and S_1. The part that is on the right of this segment is denoted as P_L^r, and the part on the left is denoted as P_L^l. The line-splitter rotates in counterclockwise order with the splitter tail being fixed. The dashed lines show the next positions the splitter will take.

Fig. IV-1(b) shows the first case that can happen. In this case, the head of the splitter points to the first encountered site, and the area on the right is greater than the area requirement of S_1. Therefore, to satisfy the area requirement, the line splitter gets rotated counterclockwise with the head being fixed. The direction is shown by a dashed line.

Fig. IV-1(c) shows the second case that can happen. In this case, the head of the splitter points to the last encountered site and the area on the right is less than the area requirement of the sites $S_{1..3}$. Therefore, to satisfy the area requirement, the line splitter gets rotated clockwise with the

head being fixed. The direction is shown by a dashed line.

Finally, Fig. IV-1(d) shows the last case. In this case, moving the head of the splitter from one position to another makes the area too large to satisfy the requirement which implies that there exists a point T between these positions that will satisfy the area requirement.



(a) Notation and order of rotation

(b) Case 1

(c) Case 2

(d) Case 3

Figure IV-1: *Notation and three cases from IHLC*

Fig. IV-2 shows an example partition from Hert & Lumelsky (1998). In the left figure, a convex polygon is shown with seven sites that have various area requirements. The figure on the right shows the final partition where each part assigned to the site has a corresponding area.

IV.2 Algorithm based on an analytical solution for bi-partition

IV.2.1 Theory

Next, we propose a different approach for the problem of convex polygon partition into multiple parts. PDAN performs a recursive partition where on each step a polygon is split into two parts according to the given area requirements and results in two sub-polygons with the highest possible compactness. Compared to the IHLC algorithm discussed in the previous sections, this algorithm would find the optimal solution for the task of splitting a polygon into two parts by

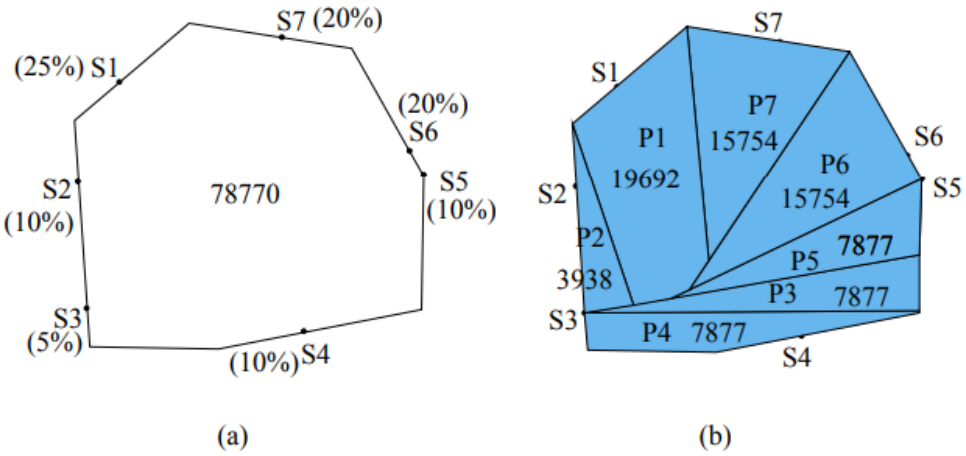


Figure IV-2: *Example partition from Hert & Lumelsky (1998).*

directly solving the equations that maximize compactness. In this algorithm, the partition is performed by straight lines. As it will be shown later, in this case, the partition into two parts can be achieved in a linear time depending on the number of vertices. Also, we do not look into partitioning by arcs in which case, as it was shown by Koutsoupias *et al.* (1992), the optimal partition can be achieved in quadratic time, $O(n^2)$, where n is the number of vertices of the polygon's border. We will call this algorithm **PDAN** which stands for **P**olygon **D**ecomposition using **AN**alytical approach.

IV.2.1.1 Problem definition

Let us define a convex polygon with perimeter P and its boundary defined by a list of n vertices V ordered in counterclockwise order. Let us also define a positive value called "area requirement" R that is less than the area of the polygon. When choosing randomly a point T on the polygon border, only one point H exists such that when the polygon is split by a line TH, the part of the polygon on the right will have the area equal to R. An example is shown in Fig. IV-3. We will name a point T as the "tail", and a point H as the "head". Let us also define a *countervetex* a point on the border of a polygon that when connected with any given point also lying on the polygon border results in two sub-polygons with one of them having the given area R. That means that any given point on the boundary always has two countervertices, with the exception of a case when R is equal to half of the polygon area. Hence, H is a countervertex of T and vice versa.

The task is to find the location of points T and H so that the corresponding right part with the area R would be the most compact. The compactness of a polygon is defined as the ratio of the square root of its area to its perimeter P. Since the area requirement is fixed, and the only variable part is the perimeter, the equation takes a form:

$$compactness = \frac{\sqrt{polygon.area}}{polygon.perimeter} = \frac{\sqrt{R}}{P(T)} \tag{IV.1}$$

where $P(T)$ is a function of the perimeter of the part on the right of the line TH. Therefore we can say that the problem of maximization of compactness is equivalent to the problem of minimization of the perimeter. As an example, the function $P(T)$ for the polygon depicted in Fig. IV-4 is shown in Fig. IV-5.

If we obtain this function, we can find its minima and the corresponding tail points T. These tail points will correspond to those lines TH that split the polygon in such a way that the parts on the right from these lines have the minimum perimeter and, correspondingly, maximum compactness.

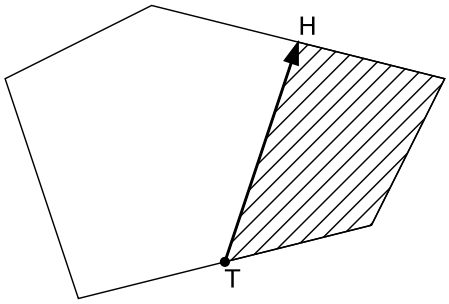


Figure IV-3: *In a convex polygon $\mathcal{P}$ with area A, for any given tail point T located on the polygon border and for any value $R \in (0, A)$, there exists only one head point H such that the area on the right of the segment TH is equal to R.*

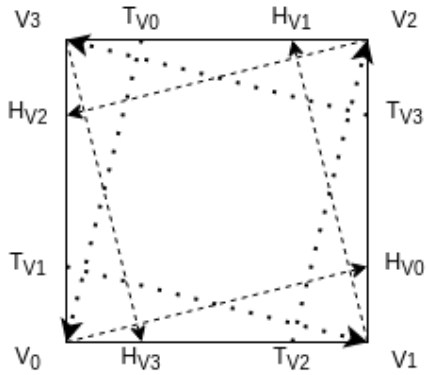


Figure IV-4: *An example polygon defined by vertices $V_0 - V_3$. For an area requirement of R=12.5%, the image shows those heads (H_{V0}, H_{V1}, H_{V2}, H_{V3}) where the corresponding tails are original vertices and vice versa (T_{V0}, T_{V1}, T_{V2}, T_{V3}). Each directed line splits the polygon into two parts. Dashed lines connect the tails lying on the original vertices with their corresponding heads. Dotted lines connect the heads lying on the original vertices with their corresponding tails.*

In order to construct this function, two tasks should be solved:

1) Find the limits of the function domains: It is clear that $P(T)$ in a general case is a piecewise-smooth function (see, for example, Fig. IV-5). The domain of each constituent function is defined by those line positions where any of the endpoints T or H moves from one segment to another. The task here is to find the limits of those domains. Our approach is presented in the next section (Section IV.2.1.2).

2) Calculate the minimum of the perimeter function $P(T)$: For each part of this piecewise function, we then need to find locations of T and H corresponding to the part with the minimum perimeter. This can be done by calculating the derivative of a function $P(T)$ for the given domain. This part is explained in Section IV.2.1.3.

It is worth noting that the number of domains does not depend only on the number of vertices. One can imagine two cases with a square polygon and two area requirements: 12.5% and 50%. The case with 12.5% is depicted in Fig. IV-4 and the corresponding function in Fig. IV-5. One can see that, in total, there are eight domains. But for the case with the area requirement equal to 50%, the number of domains would be only four as the countervertices of the original vertices of the polygon would always point to the original vertices as well.

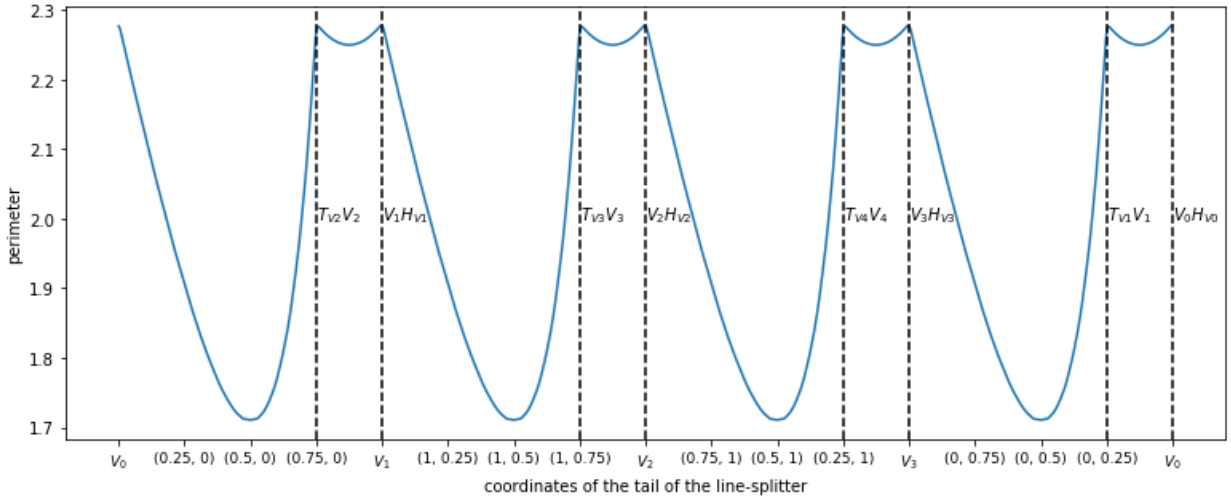


Figure IV-5: *Function $P(T)$ for the polygon given in Fig. IV-4 assuming the length of each side to be equal to 1. Vertical dashed lines correspond to the locations of the lines TH containing the original polygon vertices. Each dashed line is annotated with the vertices of the corresponding line.*

IV.2.1.2 Finding the domains

Finding the domains of the function $P(T)$ can be done in linear time $O(n)$ depending on the number of vertices n in the border of the polygon. In order to achieve that, we will take advantage of linear equations for every segment of the polygon's border as well as the shoelace formula to relate the coordinates of the vertices with the area.

The shoelace formula relates the area of a simple polygon with the coordinates of its vertices:

$$\begin{aligned} A &= \frac{1}{2}\left|\sum_{i=1}^{n-1} x_i y_{i+1} + x_n y_1 - \sum_{i=1}^{n-1} x_{i+1} y_i - x_1 y_n\right| \\ &= \frac{1}{2}\,|x_1 y_2 + x_2 y_3 + \ldots + x_{n-1} y_n + x_n y_1 \\ &\quad - x_2 y_1 - x_3 y_2 - \ldots - x_n y_{n-1} - x_1 y_n| \end{aligned} \tag{IV.2}$$

Here, A is the area of a polygon, and $x_{1..n}$ and $y_{1..n}$ are coordinates of the vertices of the polygon arranged in counterclockwise order with n being the number of vertices.

The first step of the algorithm is to choose an initial tail point T_0 on the border of the polygon. It can be any point but for simplicity, we will take one of the border vertices, V_i. Next, we want to find its corresponding countervertex. We start iterating over vertices $V_{i+2}, V_{i+3}, \ldots$ in counterclockwise order until reaching such vertex V_j so that $Area(V_i V_{i+1} \ldots V_j) \geq R$ where R is the area requirement.

In case the $Area(V_i V_{i+1} \ldots V_j)$ is equal to the area requirement R, then the head H_0 corresponding to T_0 is the last seen vertex V_j. Otherwise, we calculate R' — the difference between the area covered on the last iteration and the given area requirement, i.e.:

$$R' = Area(V_i V_{i+1} \ldots V_j) - R \tag{IV.3}$$

In such a case, the position of the head H_0 is located somewhere on the last seen segment $V_{j-1}V_j$ and we can find its exact coordinates using the shoelace formula.

Having the initial pair of tail and head vertices (T_0, H_0), we can proceed to locate the next tail T_1 that will delimit the first domain $[T_0, T_1]$ or the next head vertex H_1. Which one of them will be discovered first depends on the relative location of the vertices and the area requirement.

The next tail T_1 will be searched on the segment V_iV_{i+1} excluding the V_i as it is already taken by T_0. Likewise, the next head H_1 may be located somewhere on the segment H_0V_{j+1} excluding H_0.

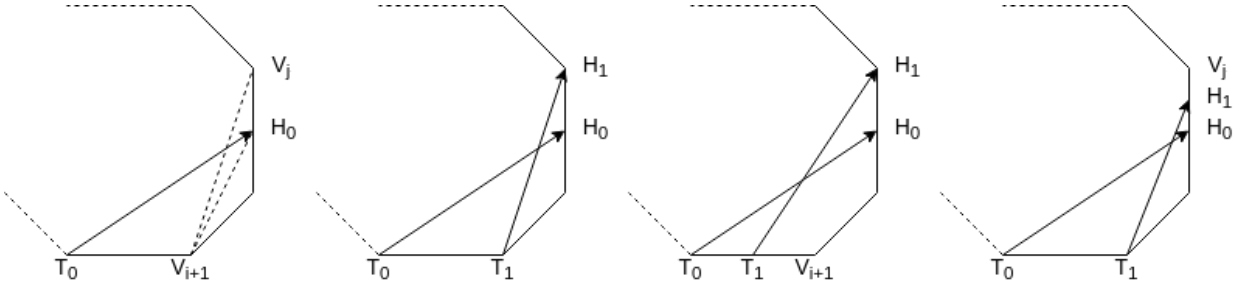


Figure IV-6: *a) T_0H_0 is the initial line-splitter where $T_0 = V_i$ is some vertex on the polygon border, V_{i+1} is the next vertex that follows after T_0, and V_j is the next vertex that follows after H_0. Comparing the areas of triangles $T_0V_{i+1}H_0$ and $V_{i+1}H_0V_j$ can tell us about the location of the next tail point T_1 delimiting the current domain and its corresponding head point H_1. b) In the case when both triangles have equal areas, the next tail and head points are V_{i+1} and V_j respectively. c) If the area of $T_0V_{i+1}H_0$ is less than the area of $V_{i+1}H_0V_j$, T_1 will be found on segment T_0V_{i+1} and $H_1 = V_j$. d) If the area of $T_0V_{i+1}H_0$ is greater than the area of $V_{i+1}H_0V_j$, H_1 will be found on segment H_0V_j and $T_1 = V_{i+1}$.*

In order to find the locations of the next head H_1 and the next tail T_1, we can compare the areas of the triangles $T_0V_{i+1}H_0$ and $V_{i+1}H_0V_j$ as shown in Fig. IV-6. Three situations can arise:

1. $Area(T_0V_{i+1}H_0) = Area(V_{i+1}H_0V_j)$. *In this case, the segment $V_{i+1}V_j$ is already a line that makes a right sub-polygon with the requested area. Therefore, T_1 will be located at V_{i+1} and the corresponding head point H_1 will be located at V_j.*

2. $Area(T_0V_{i+1}H_0) > Area(V_{i+1}H_0V_j)$. *In this case, the next head vertex H_1 will be located at V_j and we have to find T_1 somewhere on the segment T_0V_{i+1} by using the shoelace formula and the linear equation for this segment.*

3. $Area(T_0V_{i+1}H_0) < Area(V_{i+1}H_0V_j)$. *In this case, the next tail vertex T_1 will be located at V_{i+1} and its corresponding head vertex H_1 will be located on the segment H_0V_j and can be found by using the shoelace formula and the linear equation for this segment.*

Having obtained the first domain T_0T_1 and its corresponding "countersegment" H_0H_1, we can now repeat the same procedure for the pair (T_1, H_1) and obtain the next domain T_1T_2 and its corresponding countersegment H_1H_2. This process repeats until all the perimeter is covered. While iterating in this manner over the polygon's vertices, we will also get those fixed parts of the polygon between the endpoint of the domain and the first point of the countersegment. So, for example, for a domain T_kT_{k+1} and its corresponding countersegment H_kH_{k+1}, this constant area part will be delimited by vertices $T_{k+1}..H_k$ which can be either empty or not. These polygon parts will be later joined with "flexible" parts, triangles, or quadrilaterals, built on the domain segments and countersegments.

The pseudocode of the algorithm is shown in Alg. IV.2. A part of the algorithm responsible for finding the initial partition is written out separately in Alg. IV.6. The function *to_segments* returns an iterator over segments built on pairs of input vertices. The *shoelace* function takes four parameters: an area requirement and three fixed points forming a triangle. It returns a point found on the segment built on the last two points such that the area of a new triangle built on the first point, the second point, and the newly found point will be equal to the area requirement. Finally, the *remove_collinear* function takes a list of points, checks if the last three points are collinear, and removes the middle one if this is the case. Since it is easy to implement it, we do not show it here.

Algorithm IV.2: *The algorithm for calculating countersegments from a polygon's border*

Require: V — list of n vertices of polygon's border,
R — area requirement

tail_segments = to_segments($V_{0..n}$ + [V_0])
head_segments = to_segments($V_{1..n}$ + $V_{0..n}$)
D = next(tail_segments) ▷ domain segment
V_R, V_L = initial_split(heads, V, D, R) ▷ See Alg. IV.6 for details
C = Segment(V_{Rm}, V_{L0}) ▷ m - length of V_R
while D is not None **do**
    TT = Polygon(D.start, D.end, C.start) ▷ tail-based triangle
    TH = Polygon(D.end, C.start, C.end) ▷ head-based triangle
    V_R.popleft()
    V_L.append(D.start)
    if TT.area < TH.area **then**
        C.end = shoelace(TT.area, D.end, C.start, C.end)
        V_L.prepend(C.end)
        head_segments.prepend(Segment(V_{L0}, V_{L1}))
    else if TT.area > TH.area **then**
        ΔA = TT.area - TH.area
        D.end = shoelace(ΔA, C.end, D.end, D.start)
        V_R.prepend(D.end)
        TT = Polygon(D.start, D.end, C.start)
        tail_segments.prepend(Segment(V_{R0}, V_{R1}))
    yield D, C, TT.area, V_R, V_L
    V_R.popleft()
    V_R.append(C.end)
    remove_collinear(V_R) ▷ if 3 last points collinear, remove the middle one
    V_L.popleft()
    V_L.append(D.end)
    remove_collinear(V_L)
    D = next(tail_segments, on_exhaustion=None)
    C = next(head_segments)

IV.2.1.3 Finding minimum for each domain

For each domain, we will have a situation like the one presented in Fig. IV-7. Let us denote the domain segment now as S_iS_{i+1} and its countersegment as E_iE_{i+1}. We search the position of the splitter TH with the tail T between the endpoints of the domain S_iS_{i+1} and the head H between the endpoints of the corresponding countersegment E_iE_{i+1} such that the polygon $TS_{i+1}E_iH$ with area R* is the most compact. R* is the difference between the area requirement R and the area of the grey region R_F. The grey region is fixed as well as $S_{i+1}E_i$. It is also possible that the grey area will be empty.

The following lemma will help later demonstrate that finding the minimum perimeter of a polygon can be done by parts.

Lemma. *Given two functions $g(x)$ and $h(x)$ with $argmin(g(x)) = argmin(h(x)) = M$, if $f(x) = g(x) + h(x)$ then $argmin(f(x)) = M$.*

Proof. For all x: $g(x) \geq g(M) \Leftrightarrow g(x) - g(M) \geq 0$ and $h(x) \geq h(M) \Leftrightarrow h(x) - h(M) \geq 0$. And since a sum of nonnegative functions is nonnegative $\Rightarrow (g(x) - g(M)) + (h(x) - h(M)) \geq 0$ which gives $g(x) + h(x) \geq g(M) + h(M)$ and, finally, $f(x) \geq f(M)$ which corresponds to the definition

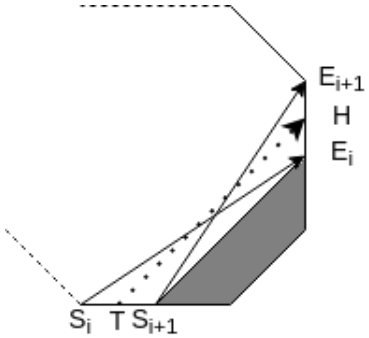


Figure IV-7: *We search for the position of the line-splitter TH where the tail point T is located on the segment S_iS_{i+1} and the head point H is located on the segment E_iE_{i+1} so that the polygon $TS_{i+1}E_iH$ with area R* is the most compact. R* is the difference of the area requirement R with the area of the grey region R_F. The grey region is fixed as well as $S_{i+1}E_i$ for each domain.*

of minimum. ■

In the following theorem, without loss of generality, we will rotate and translate the polygon such that the origin of the coordinates is fixed at the intersection of the two rays on which S_iS_{i+1} and E_iE_{i+1} are lying. Then, in the following subsections, the specific formulae will be given for the general case and for some specific cases where the formulae are undetermined.

Theorem. *Minimizing the perimeter of a quadrilateral with a fixed area, one fixed side, and fixed nonparallel rays on which the adjacent sides are located is equivalent to minimizing the total length of the adjacent sides or minimizing the length of the opposite side.*

Proof. Let us consider the example given in Fig. IV-7. We are searching for the location of points T and H so that the quadrilateral $TS_{i+1}E_iH$ has the smallest perimeter for a given area R*. The side $S_{i+1}E_i$ is fixed as well as the rays on which the segments TS_{i+1} and E_iH will be built. Let us rotate and align the figure so that the intersection of rays based on the segments S_iS_{i+1} and E_iE_{i+1} would be in $(0, 0)$ and E_iE_{i+1} would be aligned with X-axis. See Fig. IV-8. Let us also rename the vertices S_{i+1} and E_i to S and E respectively to simplify the following equations.

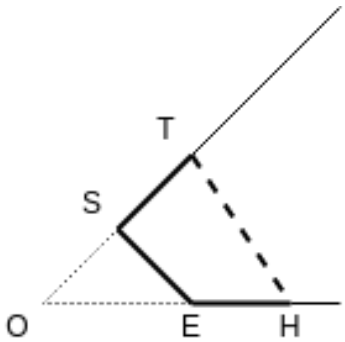


Figure IV-8: *A part of the polygon. We search for the position of the line-splitter TH such that the quadrilateral $TSEH$ with an area equal to a given area requirement is the most compact.*

From the shoelace formula (IV.2) for the given case we have:

$$2R = T_xS_y + H_xT_y - S_xT_y - E_xS_y \tag{IV.4}$$

Let $T_y = kT_x$ and $S_y = kS_x$. Then:

$$H_x = \frac{\frac{2R}{k} + E_xS_x}{T_x} \tag{IV.5}$$

We define A as $\frac{2R}{k} + E_x S_x$ to simplify the following equations:

$$H_x = \frac{A}{T_x} \tag{IV.6}$$

Let us analyze perimeters of $TS + EH = P_1$ and $TH = P_2$ separately. We assume that if both minimal solutions are equal, then this solution will correspond to the minimum of the full perimeter:

$$P = |SE| + P_1 + P_2 \tag{IV.7}$$

$$\begin{aligned} P_1 = |TS| + |EH| &= \sqrt{(T_x - S_x)^2 + (T_y - S_y)^2} \\ +H_x - E_x &= (T_x - S_x)\sqrt{1 + k^2} + \frac{A}{T_x} - E_x \end{aligned} \tag{IV.8}$$

$$\begin{aligned} P_2 = |TH| &= \sqrt{T_y^2 + (T_x - H_x)^2} \\ &= \sqrt{k^2 T_x^2 + (T_x - \frac{A}{T_x})^2} \end{aligned} \tag{IV.9}$$

Setting the derivative of P_1 equal to zero:

$$\frac{dP_1}{dT_x} = \sqrt{1 + k^2} - \frac{A}{T_x^2} = 0 \tag{IV.10}$$

gives the following T_x coordinate:

$$T_x = \pm\sqrt{\frac{A}{\sqrt{1 + k^2}}} \tag{IV.11}$$

For P_2:

$$\frac{dP_2}{dT_x} = \frac{2(T_x - \frac{A}{T_x})(1 + \frac{A}{T_x^2}) + 2k^2 T_x}{2\sqrt{k^2 T_x^2 + (T_x - \frac{A}{T_x})^2}} = 0 \tag{IV.12}$$

$$T_x - \frac{A^2}{T_x^3} + k^2 T_x = 0 \tag{IV.13}$$

$$T_x^4(1 + k^2) = A^2 \tag{IV.14}$$

we will get the same values for T_x as in Eq. (IV.11). We are interested in the positive solution only as in our case $T_x \geq S_x \geq 0$.

Both derivatives of P_1 and P_2 are negative for T_x less than the obtained solution and positive for T_x greater than the obtained solution, meaning that $argmin(P_1) = argmin(P_2)$. And as it was proved in the lemma above, it means that $argmin(P) = argmin(P_1) = argmin(P_2)$. So we can say that minimizing all the perimeter is equivalent to minimizing either of these two parts, P_1 or P_2.

Note that for the case with the negative slope of the ray based on the vertices S_i and S_{i+1} the equation will stay the same. ■

A. General solution

We have shown that minimizing the perimeter of the quadrilateral in our task is equivalent to minimizing just a part of the perimeter. We have used a simplified case where the area was rotated and translated so that we could lose some terms in the equations. Let us now solve the same task for a general case.

The shoelace formula (IV.2) gives:

$$\begin{aligned} 2R =& T_x S_y + S_x E_y + E_x H_y + H_x T_y - S_x T_y \\ & - E_x S_y - H_x E_y - T_x H_y \end{aligned} \tag{IV.15}$$

Let $T_y = k_T T_x + m_T$ and $H_y = k_H H_x + m_H$:

$$H_x = \frac{2R + E_x S_y + m_H T_x + S_x(k_T T_x + m_T) - T_x S_y - S_x E_y - m_H E_x}{(T_x(k_T - k_H) + k_H E_x + m_T - E_y} \tag{IV.16}$$

Since $S_y = k_T S_x + m_T$ and $E_y = k_H E_x + m_H$:

$$H_x = \frac{2R + E_x S_x(k_T - k_H) + (E_x + S_x - T_x)(m_T - m_H)}{T_x(k_T - k_H) + (m_T - m_H)} \tag{IV.17}$$

Let $\Delta k = k_T - k_H$ and $\Delta m = m_T - m_H$:

$$H_x = \frac{2R + E_x S_x \Delta k + (E_x + S_x - T_x)\Delta m}{T_x \Delta k + \Delta m} \tag{IV.18}$$

Let $H = 2R + E_x S_x \Delta k + (E_x + S_x)\Delta m$:

$$H_x = \frac{H - T_x \Delta m}{T_x \Delta k + \Delta m} \tag{IV.19}$$

Since minimizing the perimeter is equivalent to minimizing the sum length of segments TS and EH, let us consider the part of the perimeter containing these two segments:

$$P = \sqrt{(T_x - S_x)^2 + (T_y - S_y)^2} + \sqrt{(H_x - E_x)^2 + (H_y - E_y)^2} \tag{IV.20}$$

Substituting the ordinate coordinates gives:

$$P = |T_x - S_x|\sqrt{1 + k_T^2} + |H_x - E_x|\sqrt{1 + k_H^2} \tag{IV.21}$$

Since $\frac{dP}{dT_x} = 0$:

$$\sqrt{1 + k_T^2} \pm \frac{dH_x}{dT_x}\sqrt{1 + k_H^2} = 0 \tag{IV.22}$$

From the Eq. (IV.19) we get:

$$\frac{dH_x}{dT_x} = \frac{-\Delta m(T_x \Delta k + \Delta m) - \Delta k(H - T_x \Delta m)}{(T_x \Delta k + \Delta m)^2} = -\frac{\Delta m^2 + \Delta k H}{(T_x \Delta k + \Delta m)^2} \tag{IV.23}$$

And combining the Eq.(IV.22)) and Eq.(IV.23):

$$(T_x \Delta k + \Delta m)^2 = |\Delta m^2 + \Delta k H|\sqrt{\frac{1 + k_H^2}{1 + k_T^2}} \tag{IV.24}$$

From here:

$$T_x = \frac{\pm|\Delta m^2 + \Delta k H|\sqrt{(1 + k_H^2)/(1 + k_T^2)} - \Delta m}{\Delta k} \tag{IV.25}$$

And the sign is chosen to be plus if:

$$T_y > k_H T_x + m_H \tag{IV.26}$$

and minus otherwise.

B. Domain segment parallel to countersegment Still, a particular case exists when the domain segment is parallel to its countersegment. Then we can substitute some terms in the Eq. (IV.15) with $T_y = kT_x + m_T$, $S_y = kS_x + m_T$, $H_y = kH_x + m_H$, $E_y = kE_x + m_H$ which will give:

$$H_x = \frac{2R}{m_T - m_H} + S_x + E_x - T_x \tag{IV.27}$$

The perimeter of the quadrilateral containing the SE, ST, EH, and TH segments is as follows:

$$\begin{aligned} P =& |SE| + \sqrt{1+k^2}|(T_x - S_x) + (H_x - E_x)| \\ &+ \sqrt{(T_x - H_x)^2 + (kT_x - kH_x + m_T - m_H)^2} \end{aligned} \tag{IV.28}$$

To calculate the location of T_x for the most compact partition, we calculate the derivative:

$$\frac{dP}{dT_x} = \frac{2(T_x - H_x) + 2k(kT_x - kH_x + m_T - m_H)}{\sqrt{(T_x - H_x)^2 + (kT_x - kH_x + m_T - m_H)^2}} = 0 \tag{IV.29}$$

which gives:

$$(T_x - H_x)(1 + k^2) + k(m_T - m_H) = 0 \tag{IV.30}$$

and therefore:

$$T_x = \frac{R}{m_T - m_H} + \frac{S_x + E_x}{2} - \frac{k(m_T - m_H)}{2(1+k^2)} \tag{IV.31}$$

$$H_x = \frac{R}{m_T - m_H} + \frac{S_x + E_x}{2} + \frac{k(m_T - m_H)}{2(1+k^2)} \tag{IV.32}$$

C. Both domain segment and countersegment are parallel to Y-axis In case both the domain segment and its countersegment are parallel to Y-axis, then, from IV.15, the area of the quadrilateral can be calculated as:

$$R = (S_x - E_x)\frac{T_y - S_y + H_y - E_y}{2} \tag{IV.33}$$

From there:

$$H_y = \frac{2R}{S_x - E_x} + S_y + E_y - T_y \tag{IV.34}$$

The perimeter of the quadrilateral is calculated as:

$$\begin{aligned} P =& |SE| + |(T_y - S_y) + (H_y - E_y)| \\ &+ \sqrt{(E_x - S_x)^2 + (H_y - T_y)^2} \end{aligned} \tag{IV.35}$$

And to get the location of both the tail point and the head point that correspond to the most compact area, we calculate the derivative:

$$\frac{dP}{dT_y} = \frac{-2(H_y - T_y)}{\sqrt{(E_x - S_x)^2 + (H_y - T_y)^2}} = 0 \tag{IV.36}$$

which gives us:

$$T_y = \frac{R}{S_x - E_x} + \frac{S_y + E_y}{2} = H_y \tag{IV.37}$$

D. Domain segment parallel to Y-axis In case it is only the domain segment that is parallel to Y-axis, we can take the shoelace equation shown in Eq. (IV.15) and perform substitution $H_y = kH_x + m$ and $E_y = kE_x + m$. And since $T_x = S_x$, we will get:

$$H_x = \frac{2R + E_x S_y - kS_x E_x - mE_x - S_x S_y + S_x T_y}{-kS_x - m + T_y} \tag{IV.38}$$

To simplify this equation, let $B = kS_x + m$ and $A = 2R - BE_x + S_y(E_x - S_y)$. Then the equation takes the following simple form:

$$H_x = \frac{A + S_x T_y}{T_y - B} \tag{IV.39}$$

Next, we will analyze only the part of the perimeter that consists of TS and EH:

$$P = |T_y - S_y| + |H_x - E_x|\sqrt{1 + k^2} \tag{IV.40}$$

To get the locations of the tail and head points corresponding to the most compact partition, we calculate the derivative of the perimeter function:

$$\frac{dH_x}{dT_y} = -\frac{A + BS_x}{(T_y - B)^2} \tag{IV.41}$$

$$\frac{dP}{dT_y} = \pm 1 - \frac{(A + BS_x)\sqrt{1 + k^2}}{(T_y - B)^2} = 0 \tag{IV.42}$$

which gives:

$$T_y = B \pm \sqrt{|A + BS_x|\sqrt{1 + k^2}} \tag{IV.43}$$

The sign here is chosen according to the location of S with respect to the intersection point B of the rays. If $S_y > B$ then we choose the positive solution and, on the other hand, if $S_y < B$ then we choose the negative solution.

E. Countersegment parallel to Y-axis In case it is only the countersegment that is parallel to Y-axis, we take the Eq. (IV.15) and substitute $S_y = kS_x + m$ and $T_y = kT_x + m$. And since $H_x = E_x$:

$$H_y = \frac{2R + mS_x + kS_x E_x + E_x E_y - S_x E_y - kE_x T_x - mT_x}{E_x - T_x} \tag{IV.44}$$

Let $A = 2R + mS_x + kS_x E_x + E_x E_y - S_x E_y$ and $B = -kE_x - m$, then the equation above can be rewritten like:

$$H_y = \frac{A + BT_x}{E_x - T_x} \tag{IV.45}$$

Next, we analyze the part of the perimeter consisting of TS and EH:

$$P = |H_y - E_y| + |T_x - S_x|\sqrt{1 + k^2} \tag{IV.46}$$

By taking its derivative we can find the location of the tail corresponding to the most compact area:

$$\frac{dH_y}{dT_x} = \frac{A + BE_x}{(E_x - T_x)^2} \tag{IV.47}$$

$$\frac{dP}{dT_x} = \pm\frac{A + BE_x}{(E_x - T_x)^2} \pm \sqrt{1 + k^2} = 0 \tag{IV.48}$$

$$T_x = E_x \pm \sqrt{\frac{|A + BE_x|}{\sqrt{1 + k^2}}} \tag{IV.49}$$

And the sign is chosen according to if T is on the left or the right side of E.

IV.2.2 Algorithm

Finally, finding the most compact part out of all obtained parts is trivial. The algorithm is shown in Alg. IV.3.

It uses the *to_domains* algorithm that returns an iterator over domain segments, their corresponding countersegments, area differences, and two lists containing vertices between the domain segments and their countersegments on each side.

Then, the algorithm *to_splitter* is used. It encompasses all the algorithms discussed in the previous sections for finding a minimum perimeter for each domain including special cases. It takes the domain segment, its corresponding countersegment, and an area requirement, and returns a segment connecting the domain with the countersegment in such a way that the enclosed area has the smallest perimeter.

The *add_splitter_vertices* (Alg. IV.7) is used to include the endpoints of the splitter segment to the "right" and "left" vertices depending on if the endpoints coincide with the vertices of the polygon border or not. During the course of iteration, the shortest contours are found and returned.

The algorithm *to_domains* shown in Alg. IV.4 uses the list of vertices of the polygon's border and the area requirement to return consecutive sets of domain segments, their corresponding segments, the area differences, and lists with vertices between the segments on both sides.

First, two iterators over segments of the polygon's border are created using *to_segments* function. This function simply takes a list of vertices and constructs segments out of consecutive pairs of vertices. Its implementation is trivial, hence, we are not showing it here.

The first domain segment D is taken for which an initial split is performed using the *initial_split* algorithm. This algorithm is shown in Alg. IV.6 and its purpose is to iterate over the vertices of the polygon's border and find such a vertex when connected to the starting point of the domain segment that would yield an area greater than the input area requirement. This function returns two lists of vertices on both sides of the segments — "left" and "right" vertices as we call them.

Having obtained both lists of the vertices after the initial split, the initial countersegment is set to be a segment connecting the last point of the right-side vertices V_R with the first vertex of the left-side vertices V_L. The first head point will be located somewhere on that segment.

From this point on, the main loop of the *to_domains* algorithm starts. On each iteration, two triangles are built. The first triangle, TT or a tail-based triangle, is constructed on the endpoints of the domain and the starting point of the countersegment. The other triangle, TH or a head-based triangle, is constructed on the endpoints of the countersegment and the endpoint of the domain. By comparing the areas of these triangles it can be determined if the domain and the countersegment should be recalculated to account for the area difference or if they correspond exactly to the segments obtained previously.

In order to split the polygon into multiple parts, we simply go over a list of area requirements in the specified order and detach the most compact areas one by one. This is, of course, not the most optimal solution. But the problem of finding the best order of splitting is complex and, hence, we will not go into details on how to solve it in this paper.

The function *to_splitter* shown in IV.5 simply finds which of the cases from the previous section we have for the given domain segment and its countersegment, and calls to the corresponding function.

Algorithm IV.3: *The algorithm for finding the most compact bi-partition*

Require: V — vertices of the polygon's border oriented counterclockwise,
R — area requirement

$P_{min} = \infty$ ▷ shortest perimeter
for D, C, R*, V_R, V_L in *to_domains*(V, R) **do** ▷ See Alg. IV.4 for details on *to_domains* function
  S = to_splitter(D, C, R*) ▷ find segment giving the most compact partition
  V_R, V_L = add_splitter_vertices(V_R, V_L, S, D, C) ▷ add points of S to V_R and V_L if necessary
  if length(V_R)<P_{min} **then**
    P_{min} = length(V_R)
    $C_R = V_R$ ▷ shortest contour of a part with area R
    $C_L = V_L$ ▷ corresponding remaining contour
return C_R, C_L

Algorithm IV.4: *to_domains*

Require: V — list of n vertices of the polygon's border oriented counterclockwise,
R — area requirement

tail_segments = to_segments($V_{0..n}$ + [V_0])
head_segments = to_segments($V_{1..n}$ + $V_{0..n}$)
D = next(tail_segments)
V_R, V_L = initial_split(heads, V, D, R) ▷ Alg. IV.6
C = Segment(V_{Rm}, V_{L0}) ▷ m - number of vertices in V_R
while D is not None **do**
  TT = Polygon(D.start, D.end, C.start) ▷ tail-based triangle
  TH = Polygon(D.end, C.start, C.end) ▷ head-based triangle
  V_R.popleft()
  V_L.append(D.start)
  if TT.area < TH.area **then**
    C.end = shoelace(TT.area, D.end, C.start, C.end)
    V_L.prepend(C.end)
    head_segments.prepend(Segment(V_{L0}, V_{L1}))
  else if TT.area > TH.area **then**
    ΔA = TT.area - TH.area
    D.end = shoelace(ΔA, C.end, D.end, D.start)
    V_R.prepend(D.end)
    TT = Polygon(D.start, D.end, C.start)
    tail_segments.prepend(Segment(V_{R0}, V_{R1}))
  yield D, C, TT.area, V_R, V_L
  V_R.popleft()
  V_R.append(C.end)
  remove_collinear(V_R) ▷ if 3 last points collinear, removes the middle one
  V_L.popleft()
  V_L.append(D.end)
  remove_collinear(V_L)
  D = next(tail_segments, on_exhaustion=None)
  C = next(head_segments)

Algorithm IV.5: *to_splitter*

Require: D — domain segment,
C — countersegment,
R — area requirement

if D.start.x == D.end.x & C.end.x == C.start.x **then**
  return vertical_segments_splitter(R, D, C) ▷ Section IV.2.2 — Case C.
if D.start.x == D.end.x **then**
  return vertical_domain_splitter(R, D, C) ▷ Section IV.2.2 — Case D.
if C.start.x == C.end.x **then**
  return vertical_countersegment_splitter(R, D, C) ▷ Section IV.2.2 — Case E.
m_D, b_D = slope_intercept(D.start, D.end) ▷ m — slope, b — intercept
m_C, b_C = slope_intercept(C.start, C.end)
δb = b_D - b_C
if $m_C = m_D$ **then**
  return parallel_inclined_segments_splitter(R, δb, D, b_D, C, b_C, m_D) ▷ Section IV.2.2 — Case B.
return general_case_splitter(R, δb, D, b_D, m_D, C, b_C, m_C) ▷ Section IV.2.2 — Case A.

Algorithm IV.6: *initial_split*

Require: $head\_segments$ — iterator over "head" segments,
V — vertices of the polygon's border ordered counterclockwise,
D — initial tail segment,
R — area requirement

V_R = [] ▷ a list of "right" vertices
V_L = $V_{1..n}$ + [V_0] ▷ a list of "left" vertices
A = 0 ▷ accumulated area
for C in head_segments **do**
  V_R.append(C.start)
  V_L.popleft()
  dA = Polygon(D.start, C.start, C.end).area
  A = A + dA
  if $A < R$ **then**
    continue
  else if $A == R$ **then**
    V_R.append(C.end)
    V_L.popleft()
    C = next(head_segments)
  else
    $\delta A = dA + R - A$
    C.start = shoelace(δA, D.start, C.start, C.end)
    V_R.append(C.start)
  return V_R, V_L

Algorithm IV.7: *add_splitter_vertices*

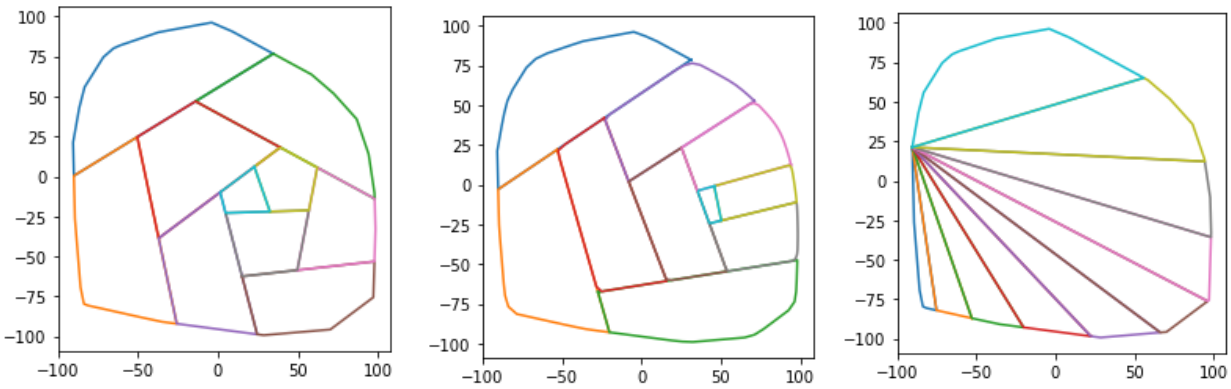
Require: V_R — vertices on the right side of the splitter,
V_R — vertices on the left side of the splitter,
D — domain segment,
C — countersegment,
S — splitter

if len(V_R) == 0 **then**
  if S.start == D.start **then**
    V_R = [D.start, D.end, C.start]
  else if S.start == D.end **then**
    V_R = [D.end, C.start, C.end]
  else
    V_R = [S.start, D.end, C.start, S.end]
else
  if D.end in Segment(S.start, V_{R_1}) **then**
    $V_R = V_{R_{1..}}$
  if C.start in Segment($V_{R_{-2}}$, S.end) **then**
    $V_R = V_{R_{..-2}}$
  V_R = [S.start] + V_R + [S.end]
if len(V_L) == 0 **then**
  if S.start == D.start **then**
    V_L = [C.start, C.end, D.start]
  else if S.start == D.end **then**
    V_L = [C.end, D.start, D.end]
  else
    V_L = [S.end, C.end, D.start, S.start]
else
  if D.start in Segment($V_{L_{-2}}$, S.start) **then**
    $V_L = V_{L_{..-2}}$
  if C.end in Segment(S.end, V_{L_1}) **then**
    $V_L = V_{R_{1..}}$
  V_L = [S.end] + V_L + [S.start]
return V_R, V_L

IV.2.3 Examples

Let us see what the partition of a polygon into ten parts looks like for a different order of splitting. Fig. IV-9 shows the partition of the same polygon into ten parts using different approaches. In this case, the area requirements were ordered from the largest to the smallest and relatively scaled as (10, 9, 8, ..., 1). We choose this order for demonstration purposes only. With any other order of area requirements or their values, the resulting partitions for each approach will be similar in terms of quality. And, as was mentioned previously, the problem of finding the best order of splitting is complex and is left out of this work.

It can be seen in Fig. IV-9(a) and Fig. IV-9(b) that both the PDAN algorithm and the approach with brute-force search result in a much better partition than what is shown in Fig. IV-9(c) built by the IHLC algorithm. It is clear that the resulting areas in Fig. (IV-9(c) are far from being compact. It can also be seen that in Fig. IV-9(b) there are some dents in the outer boundary of the polygon due to its discretization. In some cases, this can be undesirable, and the effect will be more visible



(a) PDAN (b) brute-force search (c) IHLC

Figure IV-9: *Comparison of polygon partition using three different approaches.*

for smaller numbers of vertices the border is discretized into. It is also important to note that with the discretization it is impossible to have the exact partition, so the areas slightly differ from the initial area requirements.

IV.3 Results

In this section, we show the quality of the results obtained with the presented algorithms as well as the comparison of performance.

In order to test the proposed algorithm, we used extensive tests with multiple randomly-generated polygons. To the best of our knowledge, few of the existing research works do the same.

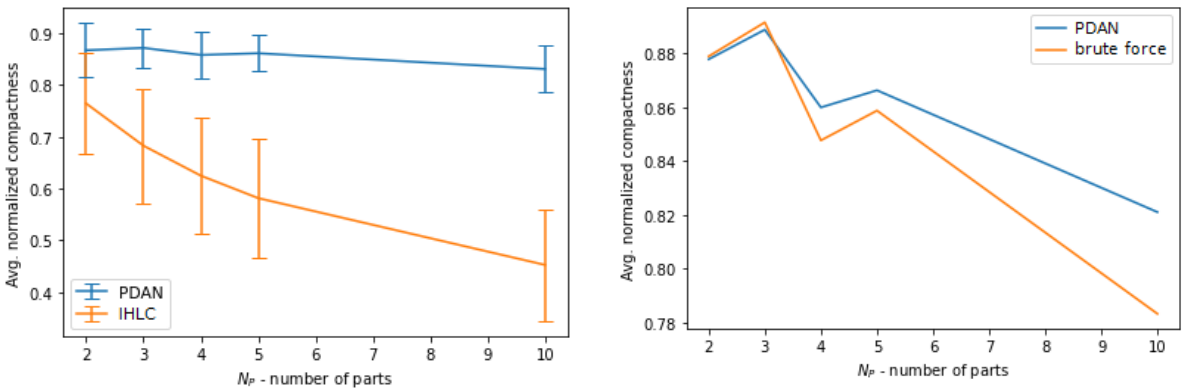
IV.3.1 Quality

The quality metrics were discussed in Section III.5. For evaluating PDAN we will compare the compactness of the resulting parts.

Fig. IV-10 shows comparisons of averaged normalized compactness with three different approaches. Normalizing compactness was done in order to get the same values for polygons with different areas but the same shapes, so that, for example, a square with an area R and a square with an area $2R$ would have the same value of compactness. We normalized the values so that the maximum possible value would be 1. And the average value was taken then over all the resulting parts of the partition.

In Fig. IV-10(a) comparison of the compactness of the PDAN algorithm with the IHLC algorithm is shown. Statistics over 100 random polygons are shown with a number of vertices ranging from five to 100. The order of splitting is chosen the same as before, starting from the largest part and ending with the smallest, where area requirements are relatively scaled as, for example (4, 3, 2, 1) for four parts. One can see that the quality of the resulting partition obtained by the IHLC algorithm is significantly worse for this setup. We can also observe the quality of the obtained result degrades with the number of parts, and more so for the IHLC algorithm.

In Fig. IV-10(b) comparison of the compactness of the proposed algorithm with the brute-force approach is shown. Here we used only one polygon, the same as in Fig. IV-9, as the brute-force search is very slow even for splitting into two or three parts. The partitions into two, three, four, five, and ten parts were calculated. The area requirements are ordered and scaled in the same



(a) PDAN vs. IHLC (b) PDAN vs. brute force

Figure IV-10: *Comparison of normalized compactness for three different approaches.*

way as was explained in the description for Fig. IV-9. For the brute-force approach, the polygon was discretized into 100 vertices. One can see that the brute-force approach results in a bit worse results than the proposed here approach. For a split into three parts, one can notice, though, that brute force yields a better result. This can be explained by the fact that it returns the parts with areas that slightly differ from the area requirements, and that due to discretization there can be some discrepancies in the shape close to the original polygon vertices.

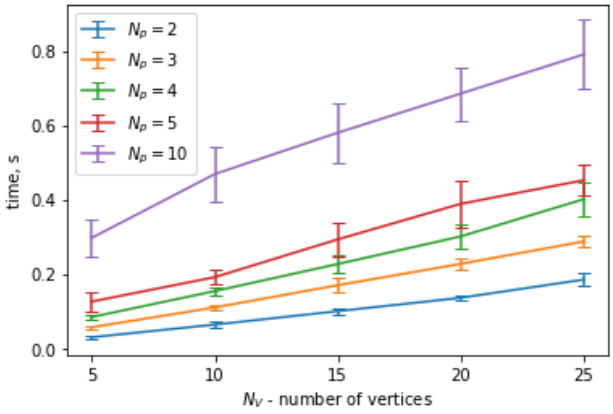
IV.3.2 Performance

Fig. IV-11 shows comparisons of the time to calculate the partition based on the number of vertices defining a polygon and the number of parts the polygon is being split into. The area requirements for each case were chosen equally. The statistics over 100 random polygons were collected for both the PDAN and the IHLC algorithms.

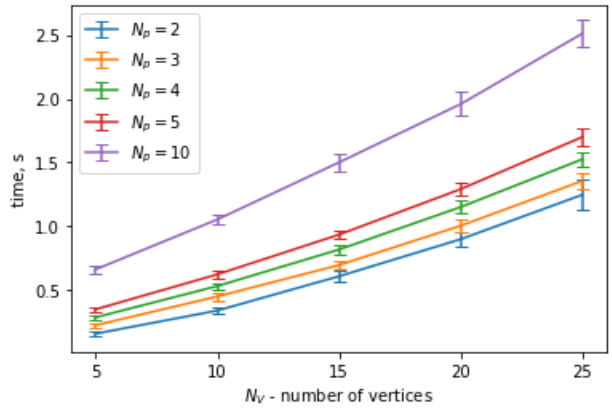
In Fig. IV-11(a) one can see the times to calculate the partitions using the algorithm described in this chapter. The times scale linearly depending on the number of vertices. It is important to note that after each iteration of finding the most compact area for a given area requirement, the coordinates of the part remaining for splitting were limited to 100 significant digits. If this was not done, the time to calculate the consequent parts could grow significantly due to the increase of significant digits after each split. So, for example, after the first iteration, if the remaining part contained a coordinate with 100 significant digits, the next iteration could yield a part with a point that had ten thousand significant digits, millions of significant digits on the next step, and so on.

Next, we check the performance of the IHLC algorithm. Fig. IV-11(b) shows that it also linearly depends on the number of vertices the polygon is built on. But the algorithm spends around three times more time on the same polygons. This comparison, though, is not entirely fair since the calculations in the implementation of the IHLC algorithm are precise, while in the implementation of PDAN we drop precision after each split. Dropping precision in the IHLC algorithm was not necessary as the times taken to calculate the partition are reasonable.

Finally, we were also interested in how our approach would scale compared to the brute-force search over a discretized polygon border. In Fig. IV-12 one can see the statistics for five random polygons. We note that it does not matter how many vertices are contained in the border of a polygon. The performance depends only on the number of vertices the polygon border is discretized into. As can be seen in the figure, the performance with this approach is much worse, even for the cases when the polygon is split into only two or three parts. This makes it barely usable in any real-life scenarios.



(a) PDAN



(b) IHLC

Figure IV-11: *Comparison of performance for the PDAN and the IHLC algorithms.*

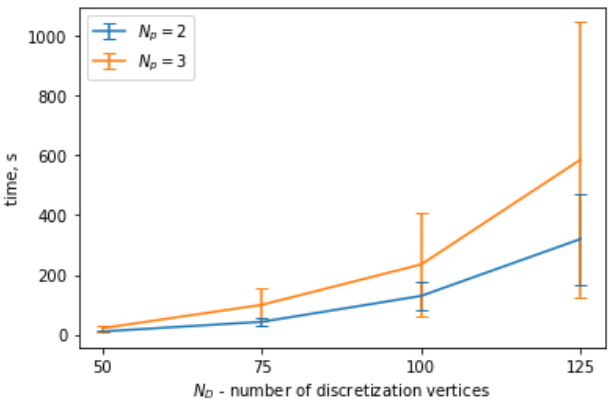


Figure IV-12: *Performance of brute-force approach depending on the number of vertices the polygon border discretized into.*

V

Non-convex Polygon Decomposition

It is clear that not every workspace that has to be covered by unmanned aerial vehicles (UAVs) can be defined by a convex polygon. In general, the area of interest can be defined by both non-convex and convex polygons and contain any number of obstacles and no-fly zones (NFZs) represented as polygon holes. Trajectory generation can both take into account the non-convex features of the area or ignore them. For example, non-convex areas can be easily closed into a convex surrounding polygon and the flight paths can be generated on this larger polygon. The extra-flight time, in this case, can be compensated by the simplicity of the flight path generated, and the sensors could be turned off when overflying parts out of the original non-convex polygon. This idea, however, is only applicable if there are no obstacles or NFZs in the surrounding convex polygon.

Hert & Lumelsky (1998) presented an algorithm for the decomposition of convex polygons which we covered in Chapter IV. This algorithm was then used by the authors as a basis for another algorithm capable of the decomposition of non-convex polygons with or without holes.

Likewise, we initially assumed that having developed an analytical algorithm that we presented in Section IV.2.1 would open a door for implementing a better algorithm for partitioning non-convex polygons as well. But after a process of trial and error designing such an algorithm, it became clear that there are several obstacles preventing making a generalization. These obstacles can be summarized as follows:

1. Maximizing compactness inside of one convex part disregards what is going to happen with the rest of the polygon. A schematic image of such an occurrence is shown in Fig. V-1. The figure shows a polygon split into three convex parts. These parts are processed one after another by the algorithm. Their areas are checked against the given area requirement. First, the algorithm visits the part on the bottom left and finds that its area satisfies the area requirement exactly. It then returns that part which is highlighted by diagonal lines. What

remains is a connected pair of convex parts where one of them is a long strip with very low compactness.

The algorithm capable of splitting a non-convex area should either be able to look ahead when detaching an area or, alternatively, it should be designed in such a way that it would be guaranteed that when detaching any selected area, the remainder would still satisfy some compactness-related criteria.

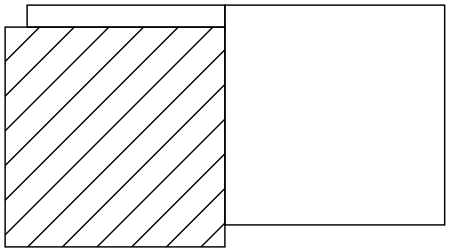


Figure V-1: *Example of algorithm's drawback.*

2. Difference in size of the parts will negatively affect the resulting partition. This is also well illustrated by Fig. V-1. If an initial partition into convex parts was to contain subpolygons homogeneous in terms of their areas, the problem could be avoided as selecting large areas is prone to leaving the remaining area of not optimal compactness.

3. There will be problems with the accumulation of multiple parts to satisfy the area requirements

   - On each accumulation of an area only a local optimum is achieved which will not guarantee the achievement of the global optimum. This is demonstrated in Fig. V-2.
   - There exists the same problem as with Hert's algorithm — to avoid discontinuous areas, some parts would be split into smaller sub-parts which will worsen the quality in many cases. An example is shown in Fig. V-3(a). If the partitioning process starts with the part on the top, it will pass through the central area which will have to be split into smaller parts in order to avoid discontinuity. A different starting point could be chosen to avoid this as shown in Fig. V-3(b).

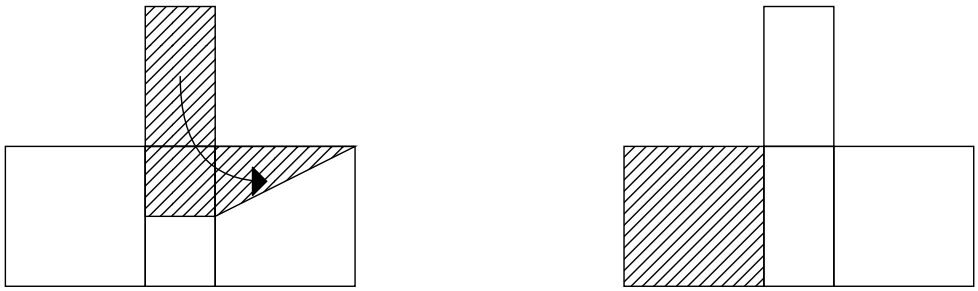
Nevertheless, the analytical algorithm for convex polygons partitioning is still useful. It produces optimal results than the algorithm it was compared against, it is efficient, and it is always possible to fall back to this very algorithm when performing partitioning of non-convex polygons when, on some step, the remaining area to be split is convex. And for the non-convex polygons, we came up with another novel algorithm that is not based on the analytical approach.

In this chapter, we propose two algorithms capable of workspace decomposition when the area is defined by non-convex polygons that can contain an undetermined number of holes inside. In Section V.1, the IHLN algorithm is presented which stands for **I**mproved algorithm by **H**ert and **L**umelsky for decomposition of **N**on-convex polygons. We discuss the original algorithm as well as our contributions that improved it. In Section V.2, a novel bottom-up algorithm is proposed based on grid decomposition. Finally, in Section V.3, results are discussed for all the proposed algorithms. We compare various metrics for the polygons produced by the algorithms as well as their performance.

(a) Accumulating a new area on each step only achieves local optimum.

(b) Alternative solution with higher compactness.

Figure V-2: *Example of the effect of the procedure of area accumulation on the final compactness.*



(a) The sub-polygon in the middle that has three neighbors is split into two parts in the course of accumulating area to satisfy an area requirement and connectivity of the resulting parts which results in poor compactness.

(b) A different starting node of the area accumulation process can result in better final compactness.

Figure V-3: *Example of good and bad partition in terms of compactness.*

V.1 Improved Hert and Lumelsky's algorithm (IHLN)

The partition of non-convex polygons is obtained by dividing the input polygon into convex parts and consecutive merging of the adjacent parts. The sweep-line algorithm is used to divide convex pieces when adding another convex part would result in a polygon with a too-large area. The sweep-line algorithm also ensures that the polygon will not become disconnected.

In Section V.1.1 we give the idea of the algorithm presented in Hert & Lumelsky (1998). In Section V.1.2 we provide an algorithm with improvements. These improvements are: we proposed a simple to-implement alternative algorithm for the decomposition of non-convex polygons into convex parts; we proposed an algorithm for constructing directed region-adjacency graph (RAG); we proposed an algorithm that unites two algorithms from Hert & Lumelsky (1998) and simplifies the logic from mutual recursive calls to a simple iteration over nodes of RAG; we proposed an improvement that removes the necessity of specifying locations of sites and presented an algorithm that assigns these locations in the course of running the algorithm.

V.1.1 Theory

The first step of the algorithm is to split an input polygon into a set of convex parts. Hert & Lumelsky (1998) provides a list of research works describing algorithms that could do that (Chazelle (1980); Greene (1983); Hertel & Mehlhorn (1983); Keil (1985); Levcopoulos & Lingas (1984); Tor & Middleditch (1984)). These algorithms were already discussed in more detail in Section II.4.

After obtaining the convex pieces, they need to be ordered in such a way that when adding the areas for each UAV with its corresponding area requirement, the remaining area of the original polygon would remain connected. Hert & Lumelsky (1998) propose an algorithm that can achieve that order. The algorithm starts at a random node and checks if it has been marked. If it hasn't been marked, it marks it and then checks if it is a leaf node (a node with no children). If it is a leaf node, it outputs the convex part corresponding to that node and then proceeds to visit all of its neighboring nodes. If this node is not a leaf node, it simply visits all of its neighboring nodes first and then outputs the corresponding convex part. The algorithm repeats this process for each unmarked node it encounters until all nodes in the graph have been marked and visited.

This algorithm operates on a RAG built on the input polygon. A RAG is a graph data structure where each node represents an area of interest of a polygon, and edges connect nodes that correspond to adjacent areas. In other words, given a polygon that has been segmented into multiple areas, a RAG represents the spatial relationships between those areas by connecting neighboring areas with edges. The resulting graph provides a way to process or analyze the structure of the polygon and to extract information about the relationships between different areas.

Hert & Lumelsky (1998) do not provide any instructions on how a RAG can be built. Therefore, we propose one way in the next section. The RAG has to contain information about parts of the input polygon as well as their connections. The algorithm iterates over the nodes of the graph and marks every node it visits. When a leaf node is encountered, it is appended to a resulting ordered list of nodes. A leaf node, in this case, is a node that has all its neighbors marked.

An example image from Hert & Lumelsky (1998) is shown in Fig. V-4. If a node $N1$ is an initial node, it is marked and now it is the turn of its neighbors to be ordered. In the case it is the node $N4$ that is chosen as a neighbor of the $N1$, it is going to be marked and next, the function will be called for the node $N5$. As the $N5$ has only one neighbor, it is returned as the first node in the resulting ordering. The next node to be considered is $N2$ as it is the next unmarked neighbor of $N4$. Since it has unmarked neighbors, the algorithm goes to the node $N3$. The node $N3$ is then returned as the second node of the ordering since all its neighbors have been already marked. Afterward, the $N2$ is returned as it doesn't have any unmarked neighbor nodes anymore. Then, the $N4$ is returned and, after it, the final node $N1$. From now on it will be assumed that the indices of the pieces correspond to the resulting ordering produced by this function.

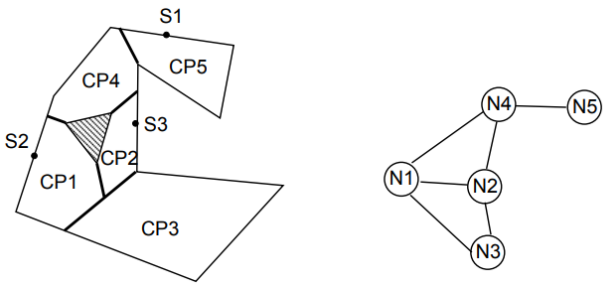


Figure V-4: *A polygon split into convex parts and its corresponding RAG.*

Next, the division process starts. The algorithm iterates over the convex parts CP_j of the input polygon and performs the following steps. For the given CP_j a polygon containing the CP_j itself plus all its predecessors reachable from CP_j without entering its successors is calculated. Authors name this polygon *PredPoly*. Examples of *PredPoly* are shown in Fig. V-5.

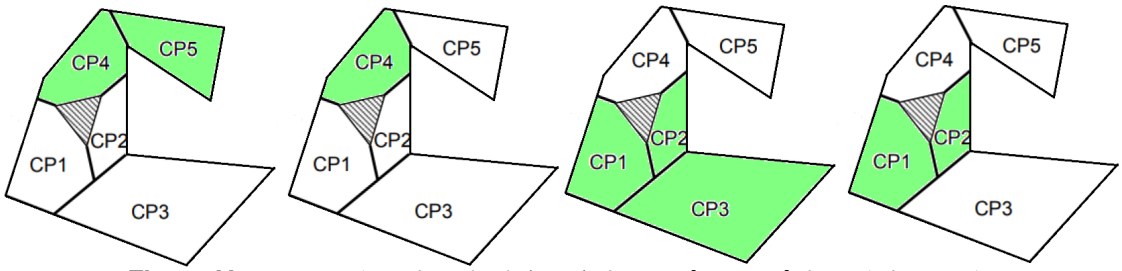


Figure V-5: *Examples of $PredPoly(CP_j)$ for j in $[5, 4, 3, 2]$ from left to right.*

Each piece CP_j is processed by dividing the corresponding *PredPoly* into parts that either will be assigned to some of the sites or stay attached to the remaining part of the original polygon. These remaining parts will be assigned later for some other $PredPoly(CP_k)$ where $k > j$. The division is proposed to be accomplished using recursion — $PredPoly(CP_j)$ is split into two parts by a polyline, one of the parts is removed completely from the original polygon and then each part gets to be divided until it is also split among the given sites. The authors propose two recursive functions to obtain the division of a convex node and its predecessor nodes. The first function, *NonconvexDivide*, shown in Alg. V.1 constructs the segments dividing *PredPoly* objects. The purpose of the second function, *DetachAndAssign*, is to remove the polygons obtained by the first function, assign them to corresponding sites, or continue their recursive division. It is shown in Alg. V.2.

V.1.1.1 *NonconvexDivide* procedure

Similarly to the Alg. IV.1, convex pieces of the input polygon will be split by line segments sweeping the areas counterclockwise. It is, however, necessary to make sure that the vertices are ordered in such a way that when the line segment is constructed to divide *PredPoly* of a given convex piece, resulting parts are not detached from the successor parts. This is done by ensuring the segment $(w_m; w_0)$ is connecting the current convex piece to its next neighbor. Vertices $w_{0..m}$ here define the border of the current convex part. And in case there is no next neighbor, the vertices are ordered such that the last vertex would be a site point.

The initial segment L is set to be (w_1, w_i) where w_i corresponds to the first site point found when iterating over the vertices in counterclockwise order starting from w_1. The w_1 point is fixed and the segment is sweeping the area by updating the other endpoint until one of the following conditions is satisfied:

1. **Case 1**. The area on the right side of L that is still not assigned to any of the sites is greater or equal to the total area requirement of all the sites on the right side (Fig. V-6);
   (a) **Case 1.1**. $Area(P^r_{L_1} + T) > AreaRequired(S(CP^r_L))$ (Fig. V-7);
   (b) **Case 1.2**. $Area(P^r_{L_1} + T) \leq AreaRequired(S(CP^r_L))$ and $Area(P^r_{L_1} + PredPoly(CP, (t_1, t_2))) < AreaRequired(S(CP^r_L))$ (Fig. V-8);
   (c) **Case 1.3**. $Area(P^r_{L_1} + T) \leq AreaRequired(S(CP^r_L))$ and $Area(P^r_{L_1} + PredPoly(CP; (t_1; t_2))) \geq AreaRequired(S(CP^r_L))$ (Fig. V-9);
2. **Case 2**. The line's endpoint has reached w_m (Fig. V-10).

In the first case either the line's endpoint does not lie on a site point and therefore the endpoint has to be moved to satisfy the area requirement, or the line's endpoint lies on a site that is also the only site at the right part of the polygon. In this latter case, it is now the turn of the starting point of a line to move counterclockwise along the border until the covered area on the right is equal to the area requirement.

Let L_1 and L_2 be segments obtained from rotating L as shown in Fig. V-6. These segments will share a point, either L_s or L_e, that does not change, and the other points are located at the vertices of the border of the polygon in such a way that

$$\begin{cases} Area(P^r_{L_1}) < AreaRequired(S(CP^r_{L_1})) \\ Area(P^r_{L_2}) > AreaRequired(S(CP^r_{L_2})) \end{cases} \tag{V.1}$$

Here, CP^r_L means the part of CP that is located on the right side of the segment L, including all the predecessors accessible from the right side. A triangle $T = (t_1, t_2, t_3)$ is defined as a difference between $CP^r_{L_1}$ and $CP^r_{L_2}$ with the segment (t_1, t_2) being the edge connecting L_1 with L_2 as shown in Fig. V-6.

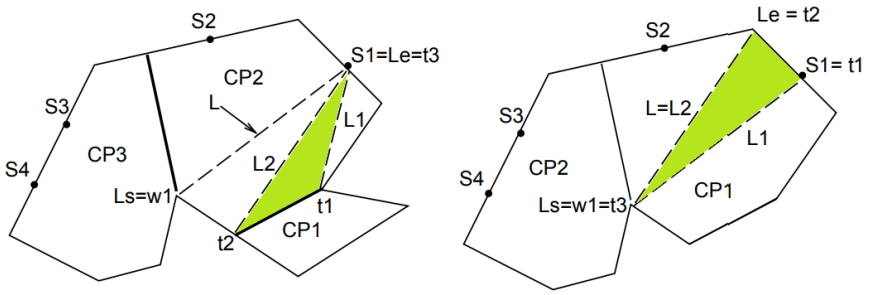


Figure V-6: *Two cases of constructing a triangle T when either L_s or L_e is fixed. T is highlighted.*

In Case 1.1, only a part of T is needed to satisfy the area requirement. A point t can be found on the segment $(t_1; t_2)$ such that

$$Area(P^r_{L_1} + T' - PredPoly(CP; (t_1; t))) = AreaRequired(S(C^r_L)) \tag{V.2}$$

where $T' = (t_1; t; t_3)$ – a triangle lying inside T which will help to satisfy the area requirement. An example is shown in Fig. V-7. There exists a point t on the segment $(w_4; w_5)$ such that AreaRequired(S_1) = Area($(w_1; w_3; w_4; t)$). After removing this polygon, CP_1 will still be attached to the remaining polygon's part. Both obtained polygons are then divided recursively.

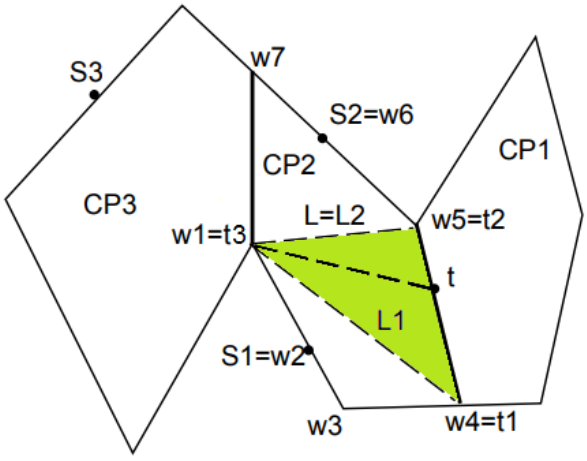


Figure V-7: *An example of Case 1.1. T is highlighted.*

In Case 1.2, all the $PredPoly(CP; (t_1; t_2))$ has to be used to satisfy the area requirement. Likewise, we can locate a point t on the segment $(t_1; t_2)$ that would give us the division of the polygon. Next, polygons $P^r_{L_1} + T'$ and $P^l_{L_1} - T' - PredPoly(CP; (t_1; t))$ are processed in a recursive manner. This case is shown in Fig. V-8.

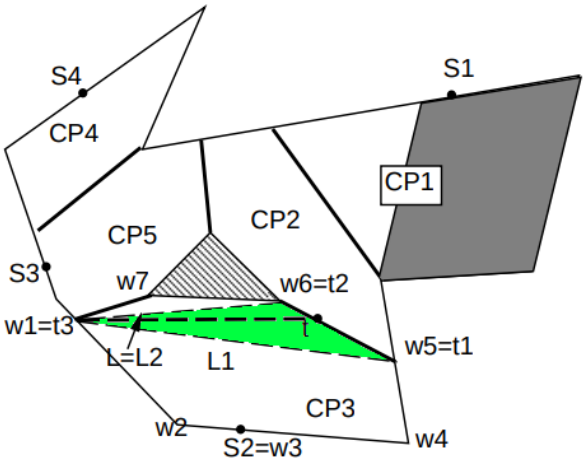


Figure V-8: *An example of the Case 1.2. T is highlighted.*

Finally, in Case 1.3, the area of the polygon $PredPoly(CP; (t1; t2))$ is greater than the sum of requirements of $S(CP_L^r)$. We need to extract only a part of the polygon that will be assigned between these sites, and the remaining part will be left to be assigned later. Here, a pseudo-site will be used to perform the division. This pseudo-site is simply a point on the edge between the current polygon part CP and its neighbor. It acts as a real site during the processing of that neighbor and later whatever area gets assigned to this pseudo-site will be reassigned back to the original site. The exact location of the pseudo-site is not specified apart from that it should be somewhere between the endpoints of the polygons' common edge (t_1, t_2). The area requirement of a pseudo-site is calculated as the difference of the sum of all the sites' area requirements found in CP_L^r and the combined area of $P_{L_1}^r$ with T' where $T' = Triangle((t_1; PS; t_3))$ and PS is the pseudo-site. Next, $PredPoly(CP; (t_1; PS))$ is partitioned and PS gets assigned its parts. Those parts are added to the union of $P_{L_1}^r$ and T' and the obtained polygon which is area-complete gets recursively divided. This case is shown in Fig. V-9.

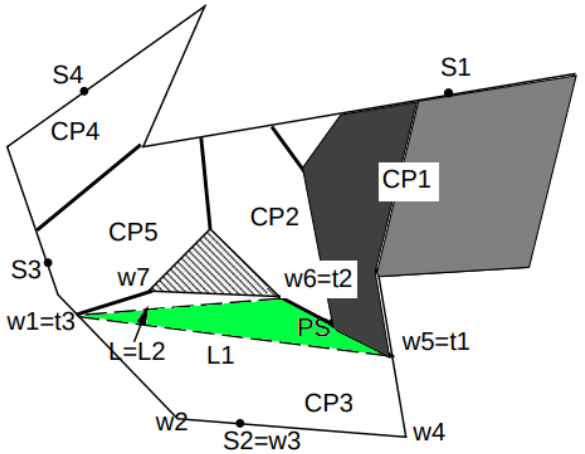


Figure V-9: *An example of the Case 1.3. T is highlighted.*

In Case 2, if sweeping the line-splitter L around the polygon does never produce the satisfactory area, pseudo-sites need to be created for each site of $S(CP_L^r)$ to get the missing areas from the successor nodes. To add a new pseudo-site, a point t is created on the segment $(w_m; w_1)$ that connects the current part with its next neighbor. Let's denote the first site encountered starting from w_1 as S_i. Then let's draw a line $L = (t; S_i)$. The vertices of P_L^r are to be ordered in such a way that $w_1 = t$. As a result, a pseudo-site is created on $(w_1; w_2)$ if the area of P_L^r is not big enough for the area requirement of S_i, or a part of P_L^r will be assigned to the site S_i to satisfy the area

requirement.

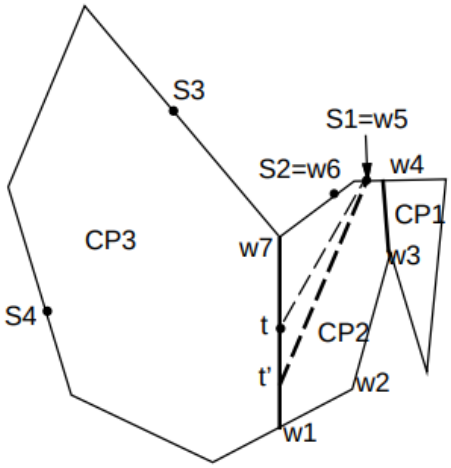


Figure V-10: *Case 2.*

In both cases, Case 1 and Case 2, when the P_L^r is divided and one of its parts gets removed, the leftover part will be divided. This implies that there is a possibility that multiple pseudo-sites will be placed on the segment $(w_m; w_1)$. $DetachAndAssign$ function is responsible for the creation of pseudo-sites and assigning polygon parts to the corresponding sites.

V.1.1.2 *DetachAndAssign* procedure

The purpose of the *DeatachAndAssign* procedure is to detach the parts of the polygon that were created by the *NonconvexDivide* algorithm and either return them as a result along with some site or call *NonconvexDivide* on it again. From now on we will refer to *NextNeighbor* as a neighbor of a node "*that is its most immediate successor*".

There are three cases that dictate how the division should happen:

1. $PredPoly(CP)$ is area-complete (the area is equal to the sum of area requirements from the sites)
2. $Area(PredPoly(CP)) < AreaRequired(S(CP))$
3. $Area(PredPoly(CP)) > AreaRequired(S(CP))$

In the first case, if the $PredPoly(CP)$ contains only one site, the $PredPoly(CP)$ can be assigned to this site, detached from the original polygon, and returned as a result. When there are multiple sites, the $PredPoly(CP)$ is also detached from the original polygon and now should be divided by the same algorithm.

In the second case, if there is only one site S_i lying in CP, all of the parts of the $PredPoly(CP)$ should be assigned to that only site, and an additional area must be taken from neighbors of CP to satisfy the area requirement. This is done by using *pseudo-sites*. A pseudo-site PS_i should be placed at the edge that connects CP to its *NextNeighbor*, and its area requirement should be the one for the previous area minus the area of $PredPoly(CP)$. After further processing of the $NextNeighbor(CP)$, the piece with the given area requirement will be assigned to the S_i. In case there are several sites lying in CP, it is necessary to divide the $PredPoly(CP)$. This division will produce two or more pieces depending on the number of sites, and one of these pieces, CP' will require more area than $PredPoly(CP)$ has. A pseudo-site will be created to connect this area with the area of the *NextNeighbor*.

In the last case, $PredPoly(CP)$ can satisfy the total area requirement of all the sites lying in CP but some area will remain unassigned and left for the sites lying outside of CP.

Algorithm V.1: *The original NonconvexDivide algorithm.*

Require: CP — a convex piece of a polygon defined by vertices $w_{1..k}$)
$L \leftarrow (w_1, w_k)$, where $w_k = S_i$ — the first site counterclockwise from w_1
$S(P_L^r) \leftarrow \{S_i\}$
while $Area(P_L^r) < AreaRequired(S(CP_L^r))$ and $L_e \neq w_m$ **do**
    if $k > 1$ and $w_{k-1} \in S(CP)$ **then**
        $S(CP_L^r) \leftarrow S(CP_L^r) \cup w_{k-1}$
    $k \leftarrow k + 1$
    $L_e \leftarrow w_k$
if $Area(P_L^r) > AreaRequired(S(CP_L^r))$ **then**
    if $L_e = S_i$ **then**
        $k_1 \leftarrow 1$
        while $Area(P_L^r) > AreaRequired(S(CP_L^r))$ **do**
            $k_1 \leftarrow k_1 + 1$
            $L_s \leftarrow w_k$
        $L_1 \leftarrow (w_{k_1}; L_e)$; $T \leftarrow (t_1; t2; t3) \leftarrow (w_{k_1}; w_{k_1} - 1; L_e)$
    else
        $L_1 \leftarrow (L_s; w_{k-1})$; $T \leftarrow (t_1; t2; t3) \leftarrow (w_{k-1}; w_k; L_s)$
    if $Area(P_{L_1}^r + T) > AreaRequired(S(CP^r))$ **then**
        Find point t on $(t1, t2)$ by interpolation
        $T' \leftarrow triangle(t_1; t; t_3)$
        $DetachAndAssign(P_{L_1}^r + T_0 - PredPoly(CP; (t1, t))$ ▷ Alg. V.2
        $DetachAndAssign(P_{L_1}^l - T')$
    else if $Area(P_{L_1}^r + PredPoly(CP; (t_1; t_2))) < AreaRequired(S(CP^r))$ **then**
        Find point t on $(t1, t2)$ by interpolation
        $T' \leftarrow triangle(t_1; t; t_3)$
        $DetachAndAssign(P_{L_1}^l + T')$
        $DetachAndAssign(P_{L_1}^r - T' - PredPoly(CP; (t_1, t))$
    else
        $PS \leftarrow apointon(t_1; t_2)$; $T' \leftarrow triangle(t_1; PS; t_3)$
        $AreaRequired(PS) \leftarrow AreaRequired(S(CP_L^r)) - Area(P_{L_1}^r + T')$
        Order $W(PredPoly(CP; (t_1; t_2)))$ such that $w_1 = PS$ if $L_e \neq S_i$ and $w_m = PS$ if $L_e = S_i$
        $DetachAndAssign(PredPoly(CP; (t1; t2)))$
        $DetachAndAssign(P_{L_1}^r + T')$
        $DetachAndAssign(P_{L_1}^l - T')$
else
    $t \leftarrow$ point on $(w_m; w_1)$
    $L_1 \leftarrow (t; S_i)$
    $DetachAndAssign(P_{L_1}^r)$
    $DetachAndAssign(P_{L_1}^l)$

Algorithm V.2: *The original DetachAndAssign algorithm*

Require: $Poly(CP)$ — polygon rooted at CP
if $||S(CP)|| = 0$ **then**
  return
if *PredPoly*(CP) is area-complete **then**
  if $S(CP) = \{S_i\}$ for some i **then**
    Assign *PredPoly(CP) to* S_i
    Detach *PredPoly(CP) from Poly(CP)*
  else
    Detach *PredPoly(CP) from Poly(CP)*
    Order $W(CP)$ so that $w_m = S_i$ for some $S_i \in S(CP)$
    NonconvexDivide(CP) ▷ Alg. V.1
else if *PredPoly*(CP) is area-incomplete **then**
  if $S(CP) = \{S_i\}$ for some i **then**
    Assign *PredPoly(CP) to* S_i
    Detach *PredPoly(CP) from Poly(CP)*
    Make pseudo-site for S_i on edge to *NextNeighbor*(CP)
  else
    Order $W(CP)$ so that edge (w_m, w_1) is the edge to *NextNeighbor*(CP)
    NonconvexDivide(CP)
else
  Order $W(CP)$ so that edge (w_m, w_1) is the edge to *NextNeighbor*(CP)
  NonconvexDivide(CP)

V.1.2 Algorithm

V.1.2.1 Decomposition into convex parts

Hert & Lumelsky (1998) provided a list of research works capable of decomposing a non-convex polygon into convex parts. We discussed all of those algorithms in Section II.4. As it was shown, there are no available implementations of algorithms capable of partitioning non-convex polygons with holes into convex ones in the public domain.

There is, of course, constrained Delaunay triangulation. But, as it will be shown later, it does not produce the best results in terms of compactness. This is why we propose an algorithm that joins triangles obtained from the regular constrained Delaunay triangulation to form larger convex parts. We present this algorithm in Alg. V.3.

The algorithm receives a polygon as input that is split into a list of triangles by applying constrained Delaunay triangulation. A random triangle is chosen and the process of accumulating its neighbors starts. The algorithm attempts to join the neighbors of the selected triangle as long as the resulting union is convex. Those neighbor triangles that would cause the union to be non-convex are ignored. Upon exhausting all the possible neighbors, the algorithm starts with the neighbors of the newly-added triangles. The process repeats until no triangles could be added to the union that would result in a convex polygon. The resulting union is returned as a result and the process repeats for the remaining part of the triangles until they are completely divided and returned. The steps of the algorithm are shown in Fig. V-11. Currently selected triangles are shown in blue and the already processed and returned parts of the polygons are shown in red.

Algorithm V.3: *The algorithm for joining together triangles obtained by Delaunay triangulation*

Require: $\mathcal{P}$ — input polygon,
E — extra_points

```
extra_constraints = [polygon.holes, E]
triangles = delaunay_triangulation(polygon.border, extra_constraints)
initial_polygon = triangles.pop()
C = []                                                    ▷ convex parts
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 ∪ 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
    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()
return C
```

V.1.2.2 RAG construction

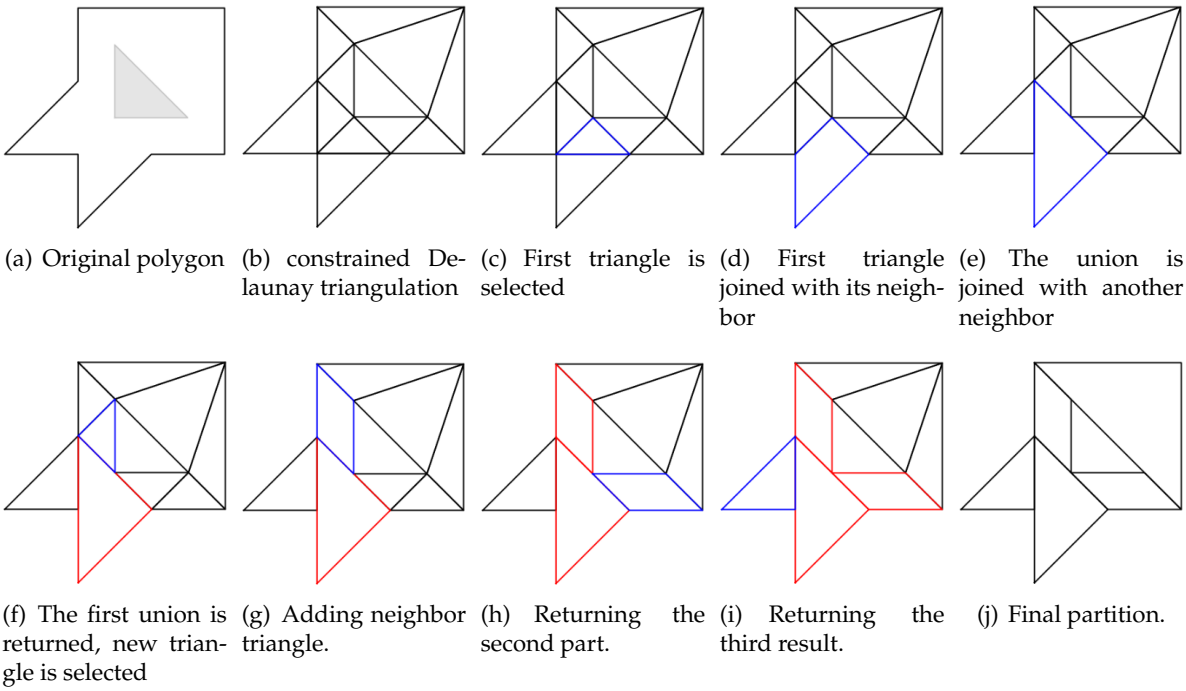
After dividing a non-convex polygon into convex parts, we need to construct a RAG out of them. We propose Alg. V.4 that can achieve that.

This algorithm simply iterates over all combinations of pairs of convex parts and adds them to an empty graph as an edge if their intersection is a segment.

Algorithm V.4: *The algorithm for constructing a RAG*

Require: P — input polygon,
S — sites,
CD — convex divisor function

```
G = Graph()                                  ▷ construct an empty graph object
EP = S - P.vertices                          ▷ extra points
CP = CD(P, EP)                               ▷ convex parts
for CP_1, CP_2 in combinations(CP) do
    intersection = CP_1 & CP_2
    if intersection is Segment then
        G.add_edge(CP_1, CP_2, intersection)
return G
```



(a) Original polygon (b) constrained Delaunay triangulation (c) First triangle is selected (d) First triangle joined with its neighbor (e) The union is joined with another neighbor

(f) The first union is returned, new triangle is selected (g) Adding neighbor triangle. (h) Returning the second part. (i) Returning the third result. (j) Final partition.

Figure V-11: *Steps of the Alg. V.3.*

V.1.2.3 Ordering nodes

The nodes of the graph then have to be ordered in such a way that when processing them, the areas would stay connected. The algorithm that was proposed in Hert & Lumelsky (1998) is, in fact, a simple post-order traversal of graph nodes.

We propose the construction of a directed RAG from the previously obtained RAG using the algorithm shown in Alg. V.5.

The algorithm takes the nodes from the original undirected graph, orders them in a post-order manner, and adds them to a new empty graph. Post-order graph traversal is a type of depth-first traversal. In binary trees it is defined as follows: 1) recursively traverse the current node's left subtree; 2) recursively traverse the current node's right subtree; 3) visit the current node.

The next step is to take the edges representing the pairs of all neighbor nodes and put them in order within each pair following the same order. These edges are then also added to the new graph.

Then it is necessary to add information about the segments between the nodes. This is done by a simple iteration over all the edges and the corresponding segments and adding them to the new graph.

The final step is to remove edges between the nodes in such a way that the resulting graph would become a binary tree. For that we iterate over each node, check if it has more than two neighbors, and if it does, we retrieve its most immediate successor, obtain the remaining neighbors of the node, and remove the edges between them. Setting an empty set of requirements for each node is done in the end.

Algorithm V.5: *The algorithm for converting an undirected graph to an ordered and directed one*

Require: UG — undirected graph
1: G = Graph() ▷ empty graph
2: ordered_nodes = UG.postorder_nodes()
3: G.add_nodes(ordered_nodes)
4: directed_edges = [sorted(e, key=ordered_nodes.index) for e in graph.edges)]
5: G.add_edges(directed_edges)
6: **for** edge, side **in** G.edges.data('side') **do**
7: **if** edge in G.edges **then**
8: G.edges[edge]['side'] = side
9: **else**
10: G.edges[edge.reversed()]['side'] = side
11: **for** node **in** G **do**
12: **if** len(G.edges[node]) < 2 **then** ▷ There can't be more than 2 outcoming edges
13: **continue**
14: most_immediate_successor = G.next_neighbor(node)
15: bad_edges_nodes = (set(G[node]) - set([most_immediate_successor]))
16: **for** neighbor **in** bad_edges_nodes **do**
17: G.remove_edge(node, neighbor)
18: **for** node **in** G **do**
19: G.nodes[node].update('requirements': frozenset())
20: **return** G

V.1.2.4 Main algorithm

The two algorithms proposed in Hert & Lumelsky (1998), *NonconvexDivide* and *DetachAndAssign* were combined into a single algorithm. Instead of relying on recursive calls, this algorithm iterates over nodes of a RAG. Since the resulting algorithm is too large to place on one page, we split it into three parts shown in Alg. V.6, Alg. V.7, and Alg. V.8.

The algorithm takes the input polygon as input, a list of *Requirement* objects, and a function for decomposing non-convex polygons into convex parts. We define a *Requirement* object as a tuple of two values, an area requirement, and an optional point. It is necessary that the input geometry objects would be of Fraction data type in order to ensure the correctness of the produced results. The sum of all the requirements has to be equal to the area of the input polygon to ensure correct results.

In Hert & Lumelsky (1998) algorithm, site assignments were always at fixed locations, but for a drone operator, the initial location of the UAVs is not a hard restriction. With very little effort the fleet can be located at any point in the area, before starting the survey flights. This constraint relaxation allows us to obtain better partitions. Thus, one of the improvements of this work proposed to improve the original algorithm is making the sites optional and letting the algorithm decide where to put them for a drone operator. The algorithm *assign_requirements* shown in Listing A-1 takes care of assigning sites and area requirements to parts of the polygons.

This algorithm checks if the input node of a graph already has a *Requirement* object assigned to it. If the node does not have any, the algorithm assigns one from the remaining requirements that do not have a location specified.

Algorithm V.6: *The IHLN algorithm*

Require: $\mathcal{P}$ — input polygon,
$R_{0..n}$ — list of n *Requirement* objects ($n > 1$),
CD — function performing decomposition into convex parts

result = [None] * len(R)
$R_{ref} = R$ ▷ reference requirements
pseudo_to_original_requirements = {}
incomplete_parts = defaultdict(list)
site_points = [r.point for r in R if r.point is not None]
G = to_graph($\mathcal{P}$, site_points, CD) ▷ graph
G = to_directed(G)
remaining_nodes = iter(G.nodes)
while G is not empty **do**
    CP = orient(next(remaining_nodes)) ▷ current polygon
    PredPolys = G.PredPolys(CP)
    CS, R = assign_requirements(CP, R, G)
    if CS is empty **then**
        continue
    required_area = sum(r.area for r in CS)
    if len(CS) = 1 and PredPolys.area < required_area **then**
        Run code from Case 1 of Alg. V.7
    else if len(CS) = 1 and PredPolys.area = required_area **then**
        Run code from Case 2 of Alg. V.8
    else
        N = G.next_neighbor(CP)
        extra_points = G.neighbor_edges_vertices(CP)
        if N is None **then**
            V = CP.border.vertices + CS + extra_points
            if CS **then**
                V = order_by_sites(V, CS_0)
        else
            e = G[CP][N]['side']
            V = CP.border.vertices + extra_points + current_sites + e.start + e.end
            V = order_by_edge(V, e)
        parts = split(CP, V, CS, G)
        G.remove_nodes_from(PredPoly)
        G = (G.prepend_two(parts, N) if len(parts) == 2 else G.prepend_three(parts, N))
    remaining_nodes = iter(G.nodes)

Algorithm V.7: *Case 1 of the IHLN algorithm*

Require: G — graph,
CP — current polygon part,
CS — sites of CP,
r — area requirement,
PredPolys — of CP,
pseudo_to_original_requirements — mapping of pseudo-sites to original sites,
incomplete_parts — a list of incomplete parts

N = G.next_neighbor(CP) ▷ neighbor
E = G[CP][N]['edge'] ▷ edge
r = CS[0]
pr = Requirement(r.area - PredPolys.area, E.centroid) ▷ pseudorequirement
PredPoly = unite(PredPolys)
original_requirement = pseudo_to_original_requirements.get(r, default=r)
incomplete_parts[original_requirement].append(PredPoly)
G.remove_nodes_from(PredPolys)
pseudo_to_original_requirements[pr] = original_requirement
G.nodes[N]['requirements'] |= pr

Algorithm V.8: *Case 2 of the IHLN algorithm*

Require: G — graph,
CS — sites of CP,
r — area requirement,
PredPolys — of CP,
pseudo_to_original_requirements — mapping of pseudo-sites to original sites,
incomplete_parts — a list of incomplete parts

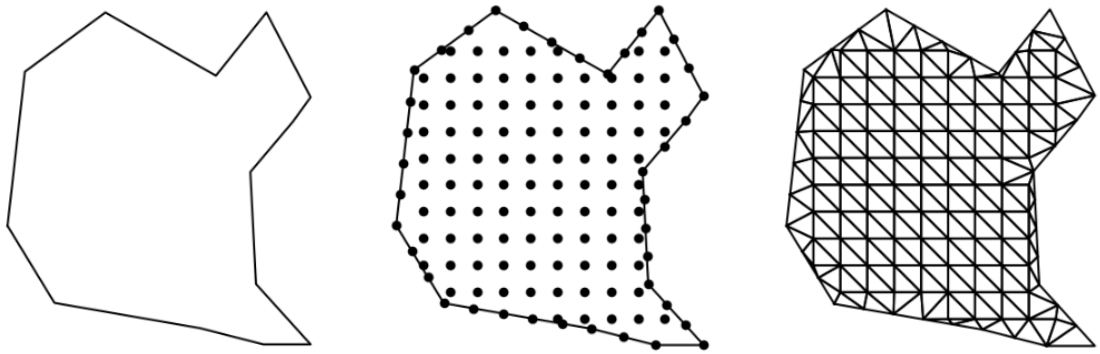
r = CS[0]
G.remove_nodes_from(PredPolys)
if r not in pseudo_to_original_requirements **then**
    resulting_part = unite(PredPolys)
else
    r = pseudo_to_original_requirements[r]
    resulting_part = unite(PredPolys, incomplete_parts[r])
found = False
for i, (r_ref, part) **in** enum(zip(R_ref, resulting_parts)) **do**
    if r == r_ref and part is None **then**
        resulting_parts[i] = resulting_part
        found = True
        break
if not found **then**
    for i, (r_ref, part) **in** enum(zip(R_ref, resulting_parts)) **do**
        if r.area == r_ref.area and part is None **then**
            resulting_parts[index] = resulting_part
            break

V.2 A bottom-up approach

In this section, we describe a completely new algorithm for the decomposition of non-convex polygons. We call it a "bottom-up" algorithm due to its logic. In this section, we explain the idea of the algorithm and provide details on how to implement it.

V.2.1 Theory

The idea of the algorithm is as follows. The algorithm receives the polygon representing the workspace (Fig. V-12(a)) and the list of area requirements for each UAV. The polygon then has to be split into a grid of convex parts. In our work, we used the constrained Delaunay triangulation with the addition of extra points across the polygon (Fig. V-12(b)). This type of triangulation is also called Delaunay triangulation with Steiner points. The result of the triangulation for the example polygon is shown in Fig. V-12(c).

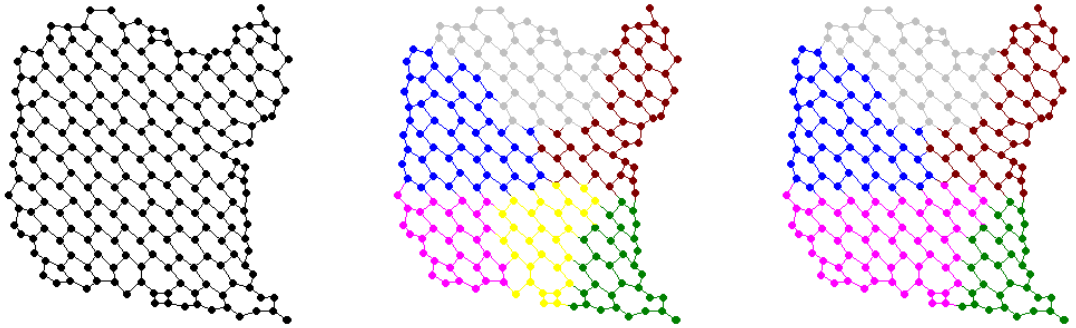


(a) Input polygon (b) Applying Steiner vertices (c) Constrained Delaunay triangulation

Figure V-12: *Partition by triangulation.*

This mesh of triangles is converted into a hierarchical RAG (Fig. V-13(a)) where each node can be either a simple polygon or another RAG. We call the nodes that are RAGs as "chunks". For simplicity, we will refer to this type of hierarchical RAG simply as RAG or just a graph.

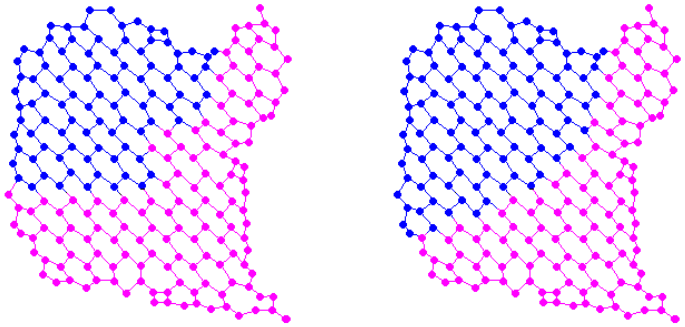
Next, an iterative process of accumulation of these chunks follows starting with those chunks with the smallest number of triangles. For each chunk, such a neighbor is chosen so that the compactness of the union would be the highest. Then the process repeats, each part is joined with a such neighbor so that their united compactness is maximized, and so on. Examples of two consecutive iterations are shown in Fig. V-13(b) and Fig. V-13(c). This process stops when one of the parts has an area close to the initial area requirement. We defined the areas as "close enough" when they are larger than half of the given area requirement. As shown later in the results, this worked perfectly in performed tests.



(a) RAG (b) RAG partition on some step N (c) RAG partition on the next step, $N+1$

Figure V-13: *Partition by triangulation.*

This process stops when one of the parts has an area close to the initial area requirement. We defined the areas as "close enough" when they are larger than half of the given area requirement. It was purely an implementation decision and, as it will be shown later in the results, this worked perfectly in performed tests. The remaining chunks that are too small are then joined together and the most compact chunk is selected to be returned (Fig. V-14(a)). The final readjustment of triangles is performed to satisfy the area requirement exactly (Fig. V-14(b)). Finally, the triangles of the resulting chunk are then joined together and returned as a result.



(a) A chunk with an area close to requirement is found (b) Result of final readjustment

Figure V-14: *Steps of the Bottom-up algorithm.*

After obtaining the part with the desired area, the process can be repeated further for other area requirements using the already computed RAG. The flowchart of the proposed algorithm is shown in Fig. V-15.

Optionally, a smoothing procedure could be added on each step for the obtained parts and the corresponding remainders. Nonetheless, in this thesis, we do not consider the problem of smoothing the borders between the polygons as some solutions have already been proposed and worked well (Wzorek *et al.*, 2021).

V.2.2 Algorithm

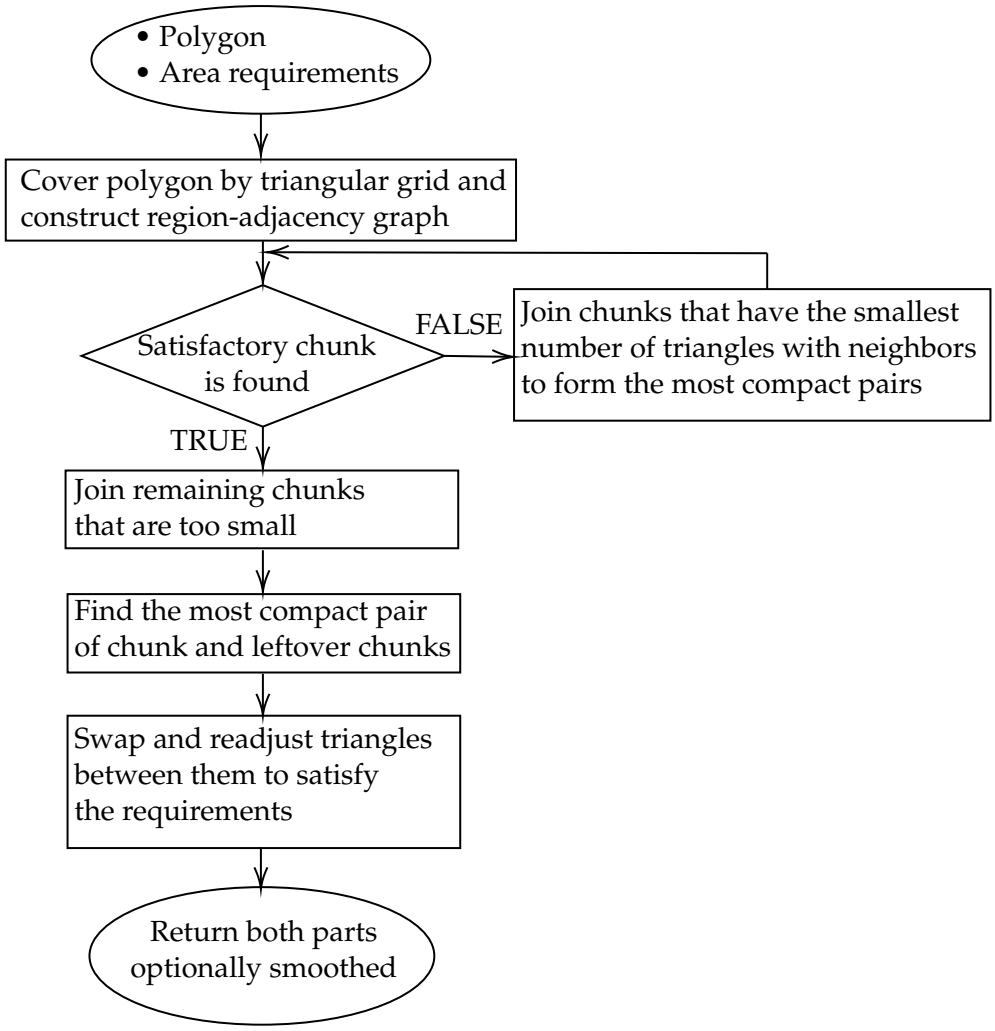


Figure V-15: *Algorithm for splitting a non-convex polygon into multiple parts according to the given area requirements*

V.2.2.1 Generating a region-adjacency graph from a polygon and a grid of points

The first step is to convert the input polygon into a RAG. In order to do that, first, we need to generate Steiner points inside the polygon. This can be done by generating a regular grid of points with a given distance between them. The points lying outside of the polygon are filtered out. At this step, a constrained Delaunay triangulation is used to obtain all the triangles that will be used as nodes of the graph. This graph will have two inner structures to keep information on both the unit triangles that we have just obtained and the collections of triangles that we will call *chunks*. These chunks will be used later when searching for a compact solution with the given area requirement.

The algorithm is shown in Alg. V.9.

Section V.2.2.2 will present the algorithm for obtaining the Steiner points. And, finally, Section V.2.2.3 shows how the region-adjacency graph can be generated.

V.2.2.2 Steiner points

There are multiple ways to place the Steiner points in the polygon. For example, as it was mentioned earlier in Chapter II, Balampanis *et al.* (2017) used Lloyd optimization (Lloyd, 1982) to have triangles of approximately the same size and the same angles close to 60 degrees. In this thesis, we

Algorithm V.9: *The algorithm for converting a polygon into a graph*

Require: $\mathcal{P}$ — input polygon,
δ — distance between Steiner points
1: extra_points = steiner_points($\mathcal{P}$, $delta$)
2: T = constrained_delaunay_triangulation($\mathcal{P}$, extra_points) ▷ triangulation object
3: triangles = T.triangles
4: triangular_map = to_triangular_map(triangles)
5: node_map = to_node_map(triangular_map)
6: G = Graph(triangular_map, node_map, $\mathcal{P}$.area)
7: **return** G

chose the simplest way by adding Steiner points at the lattice vertices of a grid constructed on top of the polygon. It is efficient and, as it will be shown later, provides a good quality of the obtained partition.

The idea of the algorithm is as follows. For a rectangle bounding the input polygon, all possible coordinates are generated with the specified distance between Steiner points. A grid of points is generated and only those points that are located within the polygon are left. Additionally, some points are placed on the border of the polygon and its inner rings delimiting the holes. Distance between the points on the segments defining those contours is chosen to be as close as possible to the original distance between Steiner points. And the original distance itself is chosen arbitrarily. In our experiments, we chose the distance to be around 1/50-1/100 part of the polygon's width.

The algorithms performing the constrained Delaunay triangulation are discussed in the literature (Sloan, 1993; Paul Chew, 1989; De Floriani & Puppo, 1992). These algorithms can either return a set of triangles lying inside the input geometry or graphs containing extra information about edges between triangles, neighbors of each triangle, and others. In the Alg:V.9 we assume that the function *constrained_delaunay_triangulation* corresponding to this algorithm returns a *Triangulation* object, a graph, containing all this extra information, but in general, any extra information is not required, since the algorithm *to_triangular_map* recalculates all the necessary connections needed for our algorithm.

V.2.2.3 Triangular and chunk maps

Next, to create a hierarchical RAG, we need to create two structures carrying information about areas inside the polygon and their common edges. The first structure is a basic RAG where the nodes are triangles that we obtained in the previous step of generating the constrained Delaunay triangulation. The second structure is also a RAG but with the nodes representing collections of triangles which we will call *chunks*. During the course of running the algorithm, these chunks will change their size by accumulating or detaching triangles or by merging with the neighbor chunks.

The Listing A-2 shows a function to create the first structure — a mapping of triangles to their neighbors by their common edges. The function iterates over all triangles and creates three mappings with edges as keys and neighbor triangles as values. In the following iterations, if another triangle has the same edge as one of the previous triangles, it will be added to that mapping and vice versa.

The Listing A-3 shows a function that creates a chunk map — a graph-like structure that contains information on the chunks, their neighbors, and their common edges. The function accepts the triangular map calculated previously and then converts all the individual triangles into chunks of size one. Then for each triangle and its neighbors, the function creates a mapping of chunks of size one created previously to dictionaries with all the edges as keys and neighbor chunks of size one as values.

V.2.2.4 Bi-partition

The algorithm for splitting a graph into two parts, shown in Alg. V.10, can be summarized as follows. On each step, it selects those chunks that consist of the smallest amount of triangles on the current and joins these chunks with those neighboring chunks that would result in the most compact union. The process is repeated until at least one chunk has an area close to the given area requirement. The definition of being "close" here is not fixed and can be tuned. Then all the remaining chunks that are too small get joined with those neighbor chunks so that the compactness of their unions would be maximized. Finally, out of the resulting chunks such chunk is chosen so that its compactness together with the compactness of the total remainder would be the highest. The remainder consisting of multiple chunks is converted into a single chunk, and the final readjustment takes place where triangles are being swapped between the chunks in order to satisfy the given area requirement precisely.

The *is_satisfactory_chunk_found* function simply iterates over all the chunks and returns *True* if at least one of them has an area close enough to the area requirement R. The closeness is defined arbitrarily. For our purposes, the condition $chunk.area \in [\frac{R}{2}; \frac{3R}{2}]$ worked fine.

Algorithm V.10: *The algorithm for splitting a graph into two parts*

Require: G — graph,
R — area requirement

```
while not is_satisfactory_chunk_found(G, R) do
    join_pairs(G)
join_small_chunks(G, R)
chunk, remainder = most_compact_pair(G)
chunk, remainder = readjust(chunk, remainder, G, R)
return chunk, remainder
```

V.2.2.5 Iterative merging of chunks

In order to merge the chunks we propose the algorithm presented in Alg. V.11.

The idea of the algorithm is to find the smallest number of triangles a chunk has and mark these chunks for joining them with other chunks. In the algorithm, this is done by collecting the chunks satisfying the condition into a set. Then, all the chunks including the ones that were just found are put into a set of potential targets for a join operation. And, as a final step, the algorithm iterates over the chunks in the first set and tries to join them with an appropriate candidate from the second set. The logic for this search and join is taken out into the Alg. V.12.

Algorithm V.11: *The algorithm for joining pairs inside the graph*

Require: G — graph

```
chunks = G.chunks
s = min(len(C.triangles) for C in chunks)                    ▷ smallest chunk size
potential_sources = OrderedSet(C for C in chunks if len(C.triangles) == s)
potetntial_targets = OrderedSet(chunks)
while potential_sources do
    merge_least_compact(G, potential_sources, potetntial_targets)    ▷ Alg. V.12
```

Algorithm V.12: *The merge_least_compact algorithm for merging least compact chunks*

Require: G — graph,
S — potential sources,
T — potential targets

source = min(S, key=compactness)
best_target = None
max_compactness = 0
N = G.neighbors(source) & T ▷ neighbors
if len(N) == 0 **then**
    S.pop(source)
    return
for neighbor **in** N **do**
    A = source.area + neighbor.area ▷ area
    P = G.union_perimeter(source, neighbor) ▷ perimeter
    compactness = $\sqrt{A}/P$
    if compactness $>$ max_compactness **then**
        best_target = neighbor
        max_compactness = compactness
S.pop(source)
S.pop(best_target)
T.pop(source)
T.pop(best_target)
G.unite(source, best_target)

V.2.2.6 Merging remaining chunks

Once the iterative join of triangles is finished the process continues with three post-processing steps. The first one consists in merging the remaining small chunks. In order to find the chunks that are not close enough to the area requirement and join them with their neighbors to form larger chunks is shown in Alg. V.13.

As the first step, the algorithm finds those chunks of the partition that do not satisfy the same condition of closeness to the area requirement as the one mentioned in Section V.2.2.4. We call these chunks "source" chunks and the chunks that can be united with "target" chunks. As long as there is at least one such source chunk and the total number of chunks in the partition is greater than two, we perform the joining process.

On each iteration of this process, we locate the source chunk with the minimum compactness and find such a target chunk that is its neighbor and gives the highest possible compactness when joined together. These chunks are joined and their union can be placed back into the set of source chunks if it still does not satisfy the aforementioned area requirement.

Algorithm V.13: *The algorithm for joining chunks*

Require: G — graph,
R — area requirement

sources = Set(chunk for chunk in G.chunks if not satisfies_area(chunk.area, R))
while len(sources) > 0 and len(G.chunks) > 2 **do**
    source = min(sources, key=compactness)
    best_target = None
    max_compactness = 0
    $\bar{N}$ = G.chunk_neighbors(source) ▷ neighbor chunks
    for N **in** $\bar{N}$ **do**
        area = source.area + N.area
        if area - R > R - source.area **then**
            continue
        perimeter = G.union_perimeter(source, N)
        $c = \sqrt{area}/perimeter$ ▷ compactness
        if c > max_compactness **then**
            best_target = N
            max_compactness = c
    sources.pop(source)
    sources.pop(best_target, None)
    if best_target is not None **then**
        union = G.unite(source, best_target)
        if not satisfies_area(union, R) **then**
            sources.add(union)

V.2.2.7 Readjustment algorithm

The readjustment algorithm is necessary to swap the triangles from one chunk to another in order to satisfy the area requirement. The algorithm that performs this readjustment is shown in Alg. V.14. It performs the reassignment of the triangles from one chunk to another until either the area requirement is satisfied or it is impossible to move a triangle without decreasing the area of a chunk below the area requirement, in which case only a part of a triangle is moved between the chunks. The algorithm orders the triangles on the border between the chunks in such a way that removing a triangle would not result in discontinuous areas.

The algorithm takes four inputs: G is the hierarchical RAG; R is the target area for each chunk; C is the chunk being adjusted; and N is the neighboring chunk being compared to C. The algorithm returns the modified chunks C and N.

The algorithm works by iteratively moving triangles from C to N, or vice versa, until the area of C is within a desired range around R. The algorithm does this by identifying a "border triangle" between C and N, which is a triangle that shares an edge with both C and N. The algorithm then moves this triangle to N and repeats the process with the new border triangle between C and N. If C becomes too small, the algorithm performs a final refinement step to ensure that C is not too small.

The algorithm also keeps track of "dismissed neighbors", which are neighboring triangles that have already been compared to C but were not suitable for adjustment. If the algorithm exhausts all border triangles without finding a suitable one to move, it resets the dismissed neighbors and tries again.

Algorithm V.14: *Neighbor chunks readjustment algorithm*

Require: G — RAG,
R — area requirement,
C — chunk,
N — neighbor chunk

```
dismissed_neighbors = OrderedSet()
while True do
    priority_triangles = filter(C.triangles, not in dismissed_neighbors)
    prioritized_triangles = priority_triangles + dismissed_neighbors
    border_triangles = filter(prioritized_triangles, touches N and not articulation triangle)
    border_triangle = next(border_triangles)
    if border_triangle is None then
        if len(dismissed_neighbors) > 1 then
            dismissed_neighbors = OrderedSet()
            continue
        divide_border_triangle(C, N, G)
        border_triangles = filter(prioritized_triangles, touches N and not articulation triangle)
    P.move(border_triangle, source=C, target=N)
    for neighbor_triangle in P.triangle_neighbors(border_triangle) do
        if neighbor_triangle in C.triangles then
            dismissed_neighbors.add(neighbor_triangle)
        if C.area == R then
            return C, N
        if C.area < R then
            G.move(border_triangle, source=N, target=C)
            final_refinement(border_triangle, R, C, N, G)
            return C, N
```

V.2.2.8 Splitting into multiple parts

Having obtained the algorithm that can split a polygon into two parts depending on the area requirements, it is trivial to extend it to split into a variable number of parts by recursively splitting parts of the polygon into two parts. The algorithm is shown in Alg. V.15.

First, in order to keep track of the area remaining after each split, a polygon called "remainder" is initialized to be equal to the input polygon. Then, the requirements are sorted from lowest to largest. For each requirement, the *split_into_two* function is called using the remainder polygon. The function returns two parts — a part that satisfies the given area requirement and the remaining part of the polygon. The remaining part is set to be a new "remainder" for a new iteration, and the part satisfying the area requirement is saved to be returned in the end when all the polygon is processed.

Algorithm V.15: *The algorithm for splitting a polygon into multiple parts*

Require: P — polygon,
R — list of area requirements,
N — approximate number of Steiner points

```
result = []
remainder = P
for R_i in sorted(R)_{..-1} do
    part, remainder = split_into_two(remainder. R_i, N)
    result.append(part)
result.append(remainder)
return result
```

V.3 Results

In this section, we present the comparison of the algorithms discussed in this chapter, the IHLN algorithm and the Bottom-up algorithm. In Section V.3.1, a comparison of the results in terms of their quality is shown. In Section V.3.2, a comparison of the results in terms of the performance of their implementations is shown.

The comparison was performed using the same set of 100 randomly-generated polygons having from three to 50 vertices and up to four holes. The polygons were split among two to ten UAVs. The area requirements for each UAV were chosen to be equal. When applicable, the number of Steiner points covering the bounding box of each polygon was chosen to be 100 which would correspond to a cell of a size of around 1/250 of the size of the original polygon.

V.3.1 Quality

The quality of the results produced by the algorithms is analyzed using the metrics given in Section III.5. First, the results for the IHLN algorithm are given in Section V.3.1.2. The results produced by the Bottom-up algorithm are given in Section V.3.1.3.

V.3.1.1 Proposed approaches

The IHLN algorithm has been extensively evaluated for four different approaches. These approaches are named as approach "A", "B", "C", and "D". Approach "A" is based on the pure Delaunay triangulation. The initial positions of the UAVs are chosen to be fixed. In approach "B", the initial positions are also fixed, but the triangles obtained from the Delaunay triangulation are joined together to decrease the total number of convex parts. The algorithm for joining the triangles was previously shown in Alg. V.3. In approach "C", the initial locations of the UAVs are not provided and the initial partition is done by the Delaunay triangulation. Finally, approach "D" is same as the approach "C" but with triangles joined together to decrease the number of convex parts. Table V-1 shows the summary of all four evaluated variants of the algorithm.

V.3.1.2 IHLN algorithm evaluation

Times to cover the areas are calculated considering the speed of the UAVs to be equal to 10m/s. It can be observed in Fig. V-16 that the compactness for all four approaches lies between 0.3 and 0.6, having still some margin for improvement. Cases "A" and "C" exhibit higher variability, and cases "C" and "D" have higher mean. Approach "D" shows the best results out of the four. A

Approach	Convex partition	UAV start
A	Delaunay triangulation	Fixed
B	Joined triangles	Fixed
C	Delaunay triangulation	Flexible
D	Joined triangles	Flexible

Table V-1: *Algorithm approaches*

similar conclusion can be deduced from Fig. V-17, and Fig. V-18 which ratifies our assumption that compactness is a good metric to anticipate the quality of the flight trajectories.

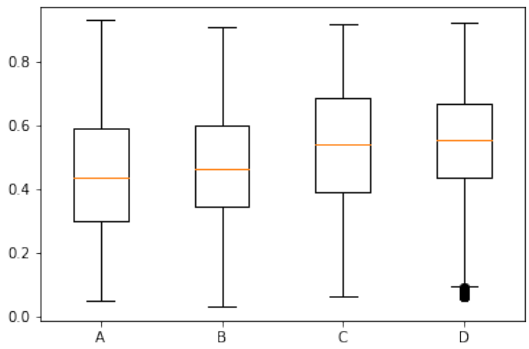


Figure V-16: *Compactness for four different approaches, "A"—"D", and 100 randomly generated polygons.*

For each metric, we also show the statistics regarding the number of UAVs (Figs. V-19, V-20, and V-21). 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 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 sub-polygons and shorter flight times. Notice from Fig. V-20 that the flight time is almost constant in approach "D" across the different numbers of UAVs. Since the number of UAVs is directly proportional to the area of the polygon, we can conclude that approach "D" scales perfectly with the area.

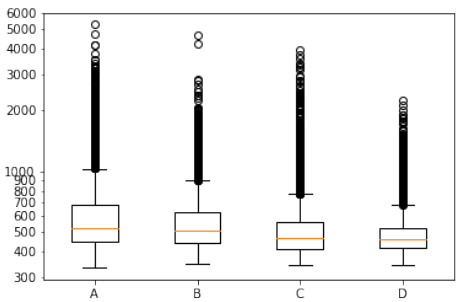


Figure V-17: *Time of flight for four different approaches, A–D, and 100 randomly generated polygons.*

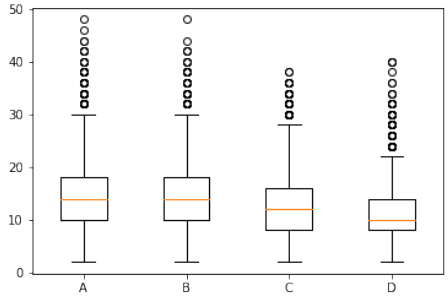


Figure V-18: *Number of turns for four different approaches, A–D, and 100 randomly generated polygons.*

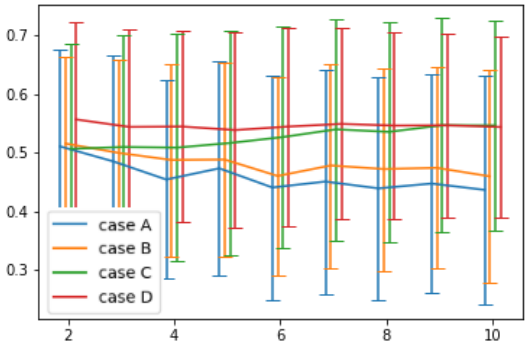


Figure V-19: *Comparison of compactness vs number of UAVs for four different cases.*

Finally, Fig. V-22 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 real-life scenarios the number of vertices will be greater than a couple of tens, and thus approach "D" would be the most appropriate.

As we can see, the compactness of polygon parts affects the trajectories of the UAVs built over these parts. This finding is in agreement with the research performed by (Wzorek *et al.*, 2021) where it was shown that maximizing the compactness of sub-polygons influences the optimality of generated motion plans.

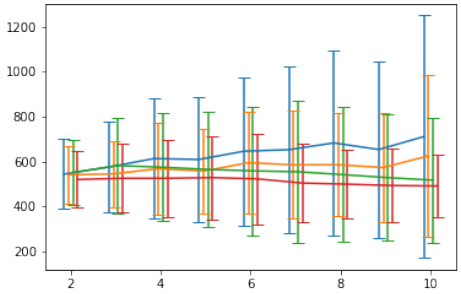


Figure V-20: *Time vs number of UAVs. Blue – case A, orange – case B, green – case C, red – case D.*

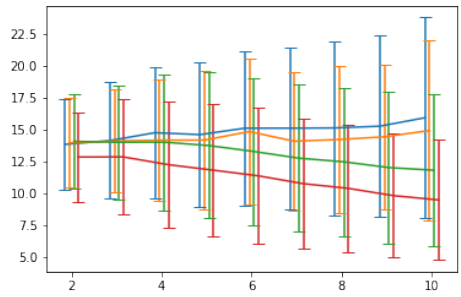


Figure V-21: *Turns vs number of UAVs. Blue – case A, orange – case B, green – case C, red – case D.*

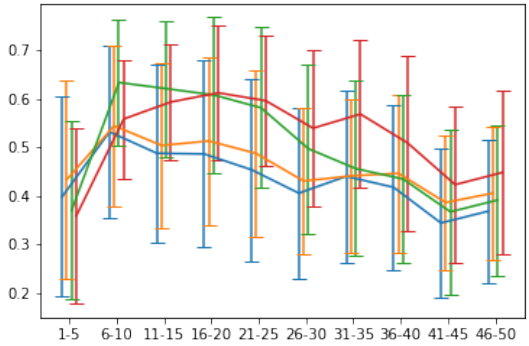


Figure V-22: *Compactness vs number of vertices of the polygon. Blue – case A, orange – case B, green – case C, red – case D.*

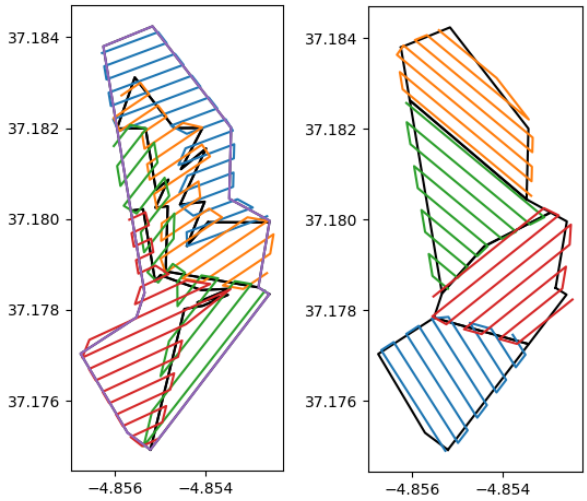


Figure V-23: *Comparison of workspace decomposition performed by IHLN and trajectory assignment using Delaunay triangulation on the left and an algorithm for joining triangles on the right.*

In Fig. V-23, one can see workspace decomposition together with the assigned trajectories. Results for two approaches are shown – one with Delaunay triangulation used for getting convex parts, and another one where the same triangles were joined together to form larger parts. The latter approach, as was mentioned before, and can be seen here, produces better results in all the metrics.

From this point on, when referring to the results obtained from the approach "D" of the IHLN algorithm, we will refer to it as simply as the IHLN algorithm.

V.3.1.3 Bottom-up

For the purpose of comparing the quality of the results produced by the presented algorithm, we used the implementation of the algorithm from (Kapoutsis *et al.*, 2017) discussed in Section II.2. As the algorithm is based on cellular decomposition, we require to represent a polygon defined by a set of vertices as a set of cells on a grid covering the polygon. The number of cells can be defined by a user. Next, the algorithm requires the initial positions of the UAVs. The original paper does not go into details on how to choose these locations but just assumes that they are given by a user. For this reason, we choose these locations in a random fashion inside the polygon. For a single polygon, we generate these locations several times and take the best result in terms of compactness.

In Fig. V-24 one can see a comparison between the IHLN algorithm, bottom-up algorithm, and the DARP algorithm from (Kapoutsis *et al.*, 2017) — the only existing open-source implementation of an algorithm from the literature, as far as we know.

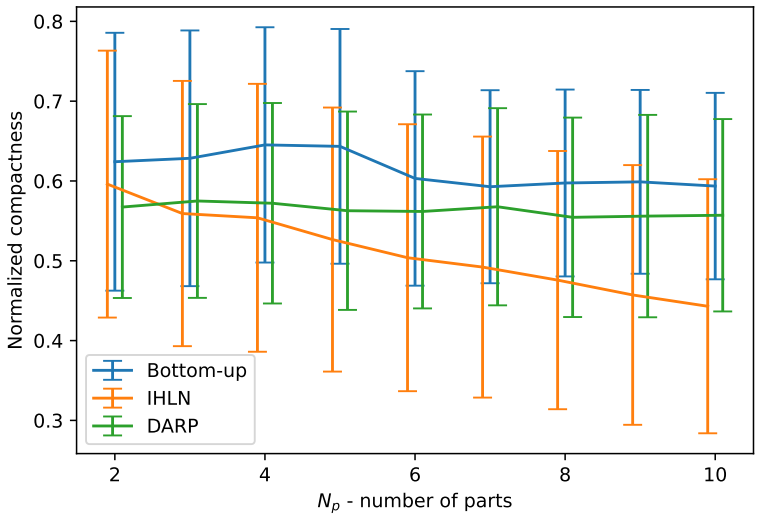


Figure V-24: *Normalized compactness vs number of parts. Comparison of IHLN algorithm, bottom-up algorithm, and the algorithm from Kapoutsis et al. (2017) — DARP.*

As can be seen, the bottom-up algorithm outperforms both IHLN and DARP in terms of compactness. From the performed analysis it is also can be seen that the IHLN algorithm does not scale well with the number of parts the polygon is split into.

Fig. V-25 shows a comparison of the number of tracks. Since the aforementioned algorithm for generating trajectories cannot deal with non-convex polygons, we used a workaround where a set of segments was generated along the line perpendicular to the width of the polygon, and intersections of the segments with the polygon were obtained. It can be seen that despite producing less compact areas, the IHLN algorithm results in fewer generated tracks.

Wzorek *et al.* (2021) performed an analysis of collective compactness scores for several configurations of their *AreaDecompose* algorithm and the algorithm of Hert & Lumelsky (1998). Their results are close to the ones achieved by us with the compactness from the algorithm by Hert & Lumelsky (1998) around 0.4 and the average compactnesses of their algorithm around the range from 0.6 to 0.65.

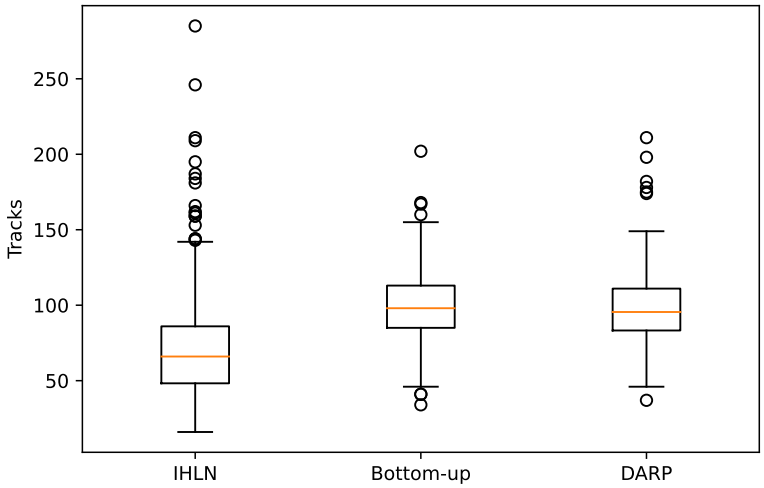
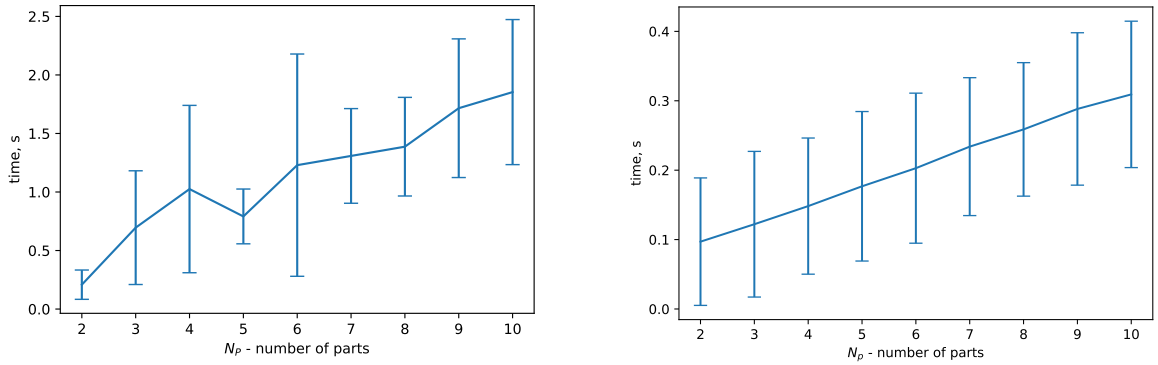


Figure V-25: *Comparison of the number of tracks.*

V.3.2 Performance

In addition to quality results, we measure the performance results, this is, the cost of obtaining the partition for each of the solutions proposed. We show the performance statistics in Fig. V-26. The figures show how execution time depends on the number of parts polygons are split into. Fig. V-26(a) shows the performance of the Bottom-up algorithm and Fig. V-26(b) shows the performance of the IHLN algorithm. One can see that the IHLN algorithm is about ten times faster but the bottom-up algorithm still performs within an acceptable time frame for a drone operator that needs to plan in advance the partition of the mission area for its fleet of UAVs.



(a) Bottom-up algorithm. (b) IHLN algorithm

Figure V-26: *Statistics for timings for 100 random polygons split into 2 to 10 parts.*

Further analysis is obtained for the Bottom-Up algorithm. Fig. V-27 shows performance times depending on the number of Steiner points. Each polygon was split into two parts and the area requirements were always equal to 50%. It can be seen that the time depends on the number of points with a non-linear function.

It is clear that the Bottom-up algorithm cannot be compared directly to the IHLN algorithm as the performance of the bottom-up's algorithm does not depend on the number of vertices of a polygon border, but on the number of Steiner points inserted inside. It does take more time, though, on average. But the gain in the compactness justifies the usage of this approach over the

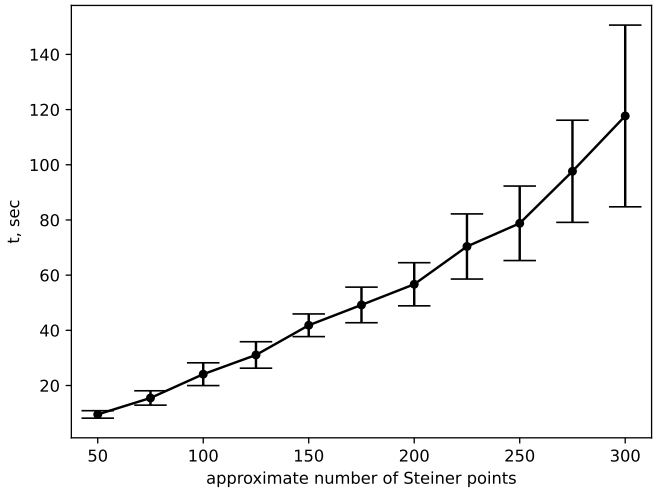


Figure V-27: *Performance of the implementation of the algorithm depending on the approximate number of Steiner points.*

other ones. And, as an option to increase the performance, one could fall back to the presented implementation of convex polygon decomposition when an input polygon is convex.

VI

Concluding Remarks

Throughout this thesis, we proposed several new algorithms and improved two existing methods for polygon decomposition, which can greatly assist drone operators in planning the flight of a heterogeneous fleet of unmanned aerial vehicles (UAVs). However, we encountered several challenges along the way, and some of these problems remain open. We provide their brief descriptions and propose them as topics for possible further research. We also present a summary of all the results achieved in the process of writing this thesis as well as the final remarks.

VI.1 Summary of Contributions

The main objective of this PhD thesis was to make a contribution in the area of workspace decomposition for multiple UAVs. We consider this goal achieved, and we think the work presented in this thesis can benefit not only operators of multiple UAVs but researchers from other areas as well where the polygon partitioning is used. Such problems can appear in various areas such as Very Large Scale Integration circuit design (Asano & Asano, 1983), parallel computing (Christou & Meyer, 1996), pattern recognition (Feng & Pavlidis, 1975), and image processing (Moitra, 1991). Likewise, operators of other types of robots rather than UAVs can benefit from the achievements of this thesis. The particular contributions of the thesis could be summarized as follows:

- In Chapter I, we presented the state of the art of the usage of fleets of drones. It was shown that effort needs to be put into designing algorithms for workspace decomposition which, in turn, justified the research work shown in this thesis
- State of the art on workspace decomposition algorithms was provided in Chapter II with

the classification of different approaches, their advantages and drawbacks as well as with information on if there are available implementations for future researchers to use them. It was shown that there are not many existing algorithms in overall and even fewer research works providing enough details for possible implementation.

- A comparison of various libraries for computational geometry was given in Chapter III which was then used to select the best-fitting library that supported all the necessary requirements for the algorithms of the area partition. We believe the analysis will be useful for other researchers in the area of computational geometry as well.
- We have provided various improvements to the algorithms initially designed by Hert & Lumelsky (1998) that are capable of workspace decomposition. It is the first open-source implementation of this algorithm[1]. The implementations of the algorithms were covered by property-based tests and the algorithms themselves ensure the correctness of the results by internally working with Fraction data types. We have also proposed using compactness as a metric to test the quality of the resulting areas. We have shown that we could benefit from an algorithm that could produce the optimal partition of non-convex polygons. We have provided a simple algorithm able to split a non-convex polygon into parts larger than triangles obtained from Delaunay triangulation by joining neighboring triangles. It was shown that in this manner the results have greater compactness than when split into triangles.
- A novel algorithm capable of decomposition of workspaces defined by convex polygons was presented in Chapter IV. The proposed algorithm is based on the analytical solution for the most compact partition of a convex polygon into two parts by a single line segment. Splitting a polygon into multiple parts was achieved by successive partitioning. It was shown that this approach yields the best results compared to other techniques when splitting convex polygons.
- Another novel algorithm for non-convex polygon decomposition was presented in Chapter V. The algorithm follows a bottom-up approach and is a cellular decomposition-based method where the cells are generated by constrained Delaunay triangulation with Steiner points. The cells are grouped together to form larger areas of high compactness. The analysis showed that this approach yields the best results compared to other algorithms for non-convex polygon decomposition
- Finally, all developed algorithms were made available and published online[2]. This is unlike the majority of other research works. All the implementations of these algorithms are also covered by property-based tests to ensure the correctness of the produced results, unlike even the most popular geometry-related libraries which are not free of issues.

VI.2 Future Research

In the process of writing this PhD thesis, various problems arose that are beyond the scope of this work. We propose these problems for potential future research.

VI.2.1 Optimal decomposition of polygons with holes

One of the extensions related to polygon decomposition is that there is no implementation of an algorithm to split a polygon with holes into a minimum number of parts. As was shown, there are

[1] https://pypi.org/project/pode/
[2] the links are not fixed by the moment of publishing the thesis; an interested user can contact the author

various algorithms that can split a simple polygon into a minimum number of parts such as the algorithm of Chazelle (1980) which does not have an open-source implementation of the algorithm of Greene (1983) which is less efficient but has an implementation in CGAL (The CGAL Project, 2022). Unfortunately, none of the algorithms works with polygons with holes which is a fairly common requirement when one requires to perform workspace decomposition. If an implementation of such an algorithm would appear, it would open the possibility of creating an algorithm for workspace decomposition that could produce even more compact solutions compared to what we achieved.

VI.2.2 Order of splitting

Another extension in the context of the algorithms proposed in Chapter IV and Chapter V is the choice of the order of the area requirements. We saw in Chapter IV that when the order is chosen to be from the largest to the smallest, the resulting partition has better quality in terms of compactness than if the order was chosen from the smallest to the largest. But it is not clear what order will result in the best possible solution in terms of compactness. What is clear is that it will be highly inefficient to search for the most optimal order of splitting. The total number of all possible ways to split a polygon into N parts when we split it into two on each step is equal to $C_{n-1}n!$. Here, C_{n-1} is a Catalan number – a number of different ways n factors can be completely parenthesized into pairs or, alternatively, the number of ways to arrange an array into a binary tree when the order does not matter. In our case, this is the number of ways to put area requirements in order for recursive bi-partition. This number is multiplied by $n!$ — a total number of permutations of an array, where array, in our case, is the array of area requirements. In this way, for two parts we can have only two possible ways to split a polygon, for three parts – 12 possible ways, for four parts – 120, for five parts – 30240, and so on. Fig. VI-1 shows that using the algorithm from Chapter IV if we search for the most optimal way to split a convex polygon into just six parts, it will take more than ten minutes to finish the calculations which is not practical.

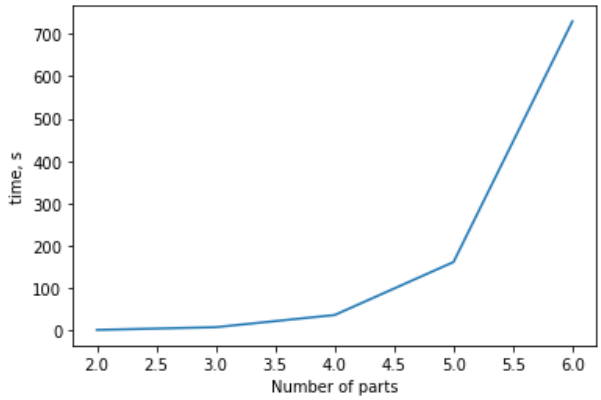


Figure VI-1: *Performance of brute-force search for the most optimal order of splitting depending on the number of parts using the algorithm for convex polygon partitioning from Chapter IV*

VI.2.3 Multipolygons

In real-life scenarios, nothing prevents the areas to be disconnected. A UAV can work first on one area and then depart to another area that is located elsewhere. In the general case, there can be multiple areas and it will be necessary to optimize either the time of flight or the distances flown between the areas.

Currently, to the best of our knowledge, there are no research works that discuss workspace

decomposition in cases like this, when the area is represented by two or more polygons. However, there exist few works that discuss coverage path planning (CPP) for disconnected areas such as Chen *et al.* (2021) and Tung & Liu (2019).

VI.2.4 Arcs

In this thesis, we only considered performing workspace decomposition using line segments to divide the polygons. In future work, algorithms using arcs could be designed, implemented, and tested. It seems that the results could improve from this change but how much of an improvement we would see can be verified only by testing.

The work by Koutsoupias *et al.* (1992) shows that to obtain the optimal bisection of a convex polygon in terms of the total perimeter can be achieved in quadratic time using arcs. Applying the proposed solution recursively in order to divide a polygon into multiple parts could result in more compact shapes.

VI.2.5 Random polygons generation

The current work relies on the generation of random polygons for the purpose of evaluating the effectiveness of the algorithms and testing the correctness of the implementations. As was mentioned in Section III.3, an algorithm implemented in the *hypothesis-geometry* library (Ibrakov, 2022) was used. That algorithm generates the polygons by trial and error but more effective ways may exist.

There exist multiple algorithms in the literature capable of generating random polygons but, to the best of our knowledge, there are no works analyzing their differences. Moreover, most of the algorithms in the literature do not have freely available implementations. A proper evaluation of their effectiveness and their differences would benefit future researchers in the whole area of computational geometry.

VI.2.6 Exploring different metrics for analytical partitioning

In Chapter IV, we proposed an algorithm capable of partitioning a convex polygon based on the compactness metric. However, there could be other metrics that could be used to obtain the partition.

One could come up with various metrics such as aspect ratio which would measure the ratio of the longest side of a polygon to its shortest side, or the ratio of the area of the workspace to the area of the smallest bounding box that encloses it.

We can only speculate how the choice of different metrics could affect the resulting areas. And the proposed algorithm that works with compactness would need to be updated to work with other metrics.

VI.2.7 Edge smoothing

As it was shown, the algorithm proposed in Chapter V works well for the specific objective of area partition for a later assignment of trajectories over the obtained parts. However, for other purposes, smoother borders between the parts could be necessary.

There exist various algorithms for curve smoothing. For example, the most well-known Ramer–Douglas–Peucker algorithm (Ramer, 1972) could be used to reduce the complexity of zig-zag lines that appear due to the nature of the algorithm (an example is shown in Fig. VI-2). This

and similar algorithms, however, do not preserve the areas on the sides of the lines. In many practical cases, this will not be a problem, especially for the cases when the input polygon has to be split into a few parts of relatively similar sizes. However, in a general case, an algorithm that can ensure the correctness of the results will be necessary. One such algorithm that ensures the conservation of areas is Kronenfeld *et al.* (2020). This algorithm has an open-source implementation provided by the authors themselves[3].

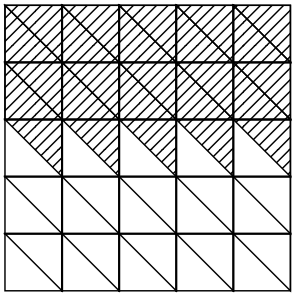


Figure VI-2: *An example of the Bottom-up algorithm's drawback — grouping triangles can result in zigzag lines. An algorithm to smooth them is necessary to avoid this.*

VI.2.8 Addressing overlapping subspaces

One important aspect to address is the assumption of non-overlapping subspaces, which may not align with the requirements of actual drone operations. Many applications require overlapping areas flown with different aircraft and sensors, posing engineering challenges beyond computational geometry. Therefore, it is necessary to explore optimal flight planning for simultaneous operations of fleets of drones taking into account the potential overlapping subspaces.

To tackle these challenges, novel approaches should be investigated for coordinating multiple UAVs within shared workspaces. Conducting a comprehensive analysis of the impact of overlapping subspaces on mission performance is essential. This analysis should evaluate the effects on various metrics such as mission effectiveness, data quality, and operational costs in various application scenarios such as surveillance, search and rescue, precision agriculture, and others.

[3] https://github.com/geobarry/line-simplify

A

Appendix

Listing A-1: Function assigning requirements to nodes.

```
def assign_requirements(polygon: Polygon, requirements: list[Requirement],
                        graph: Graph) -> tuple[FrozenSet[Requirement], list[
    Requirement]]:
    """
    :param polygon: a polygon part corresponding to some node in the graph
    :param requirements: a list of all not yet assigned requirements
    :param graph: RAG
    :returns: requirements assigned to the input polgygon and the remaining
    requirements
    """
    requirements_with_points = {requirement for requirement in requirements
                                if requirement.point is not None}
    preassigned_requirements_with_points = {
        site for node in graph for site in graph.nodes[node]['requirements']}
    if (not requirements_with_points and not
    preassigned_requirements_with_points):
        bare_requirements = [requirement for requirement in requirements
                             if requirement not in requirements_with_points]
        if not bare_requirements:
            return frozenset({}), requirements
        *requirements, requirement = requirements
        point = polygon.border.vertices[0]
        requirement = Requirement(requirement.area, point=point)
        return frozenset({requirement}), requirements
    current_preassigned_requirements = graph.nodes[polygon]['requirements']
    ancestors = nx.ancestors(graph, polygon)
    remaining_nodes = set(graph.nodes) - {polygon, *ancestors}
    if not remaining_nodes:
        leftover_requirements = [requirement for requirement in requirements
                                 if requirement not in
    requirements_with_points]
        return (current_preassigned_requirements | requirements_with_points,
    leftover_requirements)
    current_requirements = {requirement for requirement in
    requirements_with_points
                            if requirement.point in polygon}
    polygon_requirements_only = {requirement for requirement in
    current_requirements
                                 if all(requirement.point not in node for node
     in remaining_nodes)}
    if polygon_requirements_only or current_preassigned_requirements:
        requirements_to_return = frozenset(polygon_requirements_only |
    current_preassigned_requirements)
        leftover_requirements = [requirement for requirement in requirements
                                 if requirement not in requirements_to_return]
        return requirements_to_return, leftover_requirements
    if not current_requirements:
        return frozenset({}), requirements
    current_requirement = current_requirements.pop()
    leftover_requirements = [requirement for requirement in requirements
                             if requirement != current_requirement]
    return frozenset([current_requirement]), leftover_requirements
```

Listing A-2: Function to create a triangular map

```
def to_triangular_map(triangles: List[CachedPolygon]) -> TriangularMap:
    """
    :param triangles: a list of triangles
    :return: mapping storing relations between neighbouring triangles
    and shared segments
    """
    triangular_map = {}
    triangle_by_edge = {}
    for triangle in triangles:
        triangular_map[triangle] = {}
        for edge in triangle.edges:
            neighbor = triangle_by_edge.get(edge)
            triangular_map[triangle][edge] = neighbor
            if neighbor is None:
                triangle_by_edge[edge] = triangle
            else:
                triangular_map[neighbor][edge] = triangle
    return triangular_map
```

Listing A-3: Function to create a chunk map.

```
def to_chunk_map(triangular_map: TriangularMap) -> Dict[Chunk,
                                                        NeighborChunkPerEdge]:
    """
    :param triangular_map: input triangular map
    :return: same triangular map but where triangles are wrapped as
    chunks and segments wrapped as sets of single segment
    """
    chunk_per_triangle = {triangle: to_chunk(triangle)
                          for triangle in triangular_map}
    return {chunk_per_triangle[triangle]:
            {frozenset([edge]): chunk_per_triangle[neighbor]
             for edge, neighbor in neighbors_map.items()
             if neighbor is not None}
            for triangle, neighbors_map in triangular_map.items()}
```

Bibliography

ANN, SUNGJUN, KIM, YOUDAN, & AHN, JAEMYUNG. 2015. Area allocation algorithm for multiple uavs area coverage based on clustering and graph method. *Ifac-papersonline*, **48**(9), 204–209. vii, 11

APOSTOLIDIS, SAVVAS D., KAPOUTSIS, PAVLOS CH., KAPOUTSIS, ATHANASIOS CH., & KOSMATOPOULOS, ELIAS B. 2022. Cooperative multi-UAV coverage mission planning platform for remote sensing applications. *Autonomous robots*, jan. vii, 12

ASANO, TETSUO, & ASANO, TAKAO. 1983. Minimum partition of polygonal regions into trapezoids. *Pages 233–241 of: 24th annual symposium on foundations of computer science (sfcs 1983)*. IEEE. 81

ATIF, MUHAMMAD, AHMAD, RIZWAN, AHMAD, WAQAS, ZHAO, LIANG, & RODRIGUES, JOEL JPC. 2021. Uav-assisted wireless localization for search and rescue. *Ieee systems journal*, **15**(3), 3261–3272. 1

AUER, THOMAS, & HELD, MARTIN. 1996. Rpg-heuristics for the generation of random polygons. *Pages 38–44 of: Proc. 8th canada conf. comput. geom. ottawa, canada*. Citeseer. 25

BAILON-RUIZ, RAFAEL, REYMANN, CHRISTOPHE, LACROIX, SIMON, HATTENBERGER, GAUTIER, DE MARINA, HECTOR GARCIA, & LAMRAOUI, FAYÇAL. 2017. System simulation of a fleet of drones to probe cumulus clouds. *Pages 375–382 of: Unmanned aircraft systems (icuas), 2017 international conference on*. IEEE. 4

BALAMPANIS, FOTIOS, MAZA, IVÁN, & OLLERO, ANÍBAL. 2017. Coastal areas division and coverage with multiple uavs for remote sensing. *Sensors*, **17**(4), 808. 12, 68

BARRADO, CRISTINA, PASTOR, ENRIQUE, VALERO, MIGUEL, ROYO, MARIA, SALAMÍ, ESTHER, REYES, ANGÉLICA, & ROYO, PABLO. 2016. *Operaciones de vuelo para múltiples aviones remotamente pilotados*. *https://futur.upc.edu/19380062*. 4

BARRIENTOS, ANTONIO, COLORADO, JULIAN, CERRO, JAIME DEL, MARTINEZ, ALEXANDER, ROSSI, CLAUDIO, SANZ, DAVID, & VALENTE, JOAO. 2011. Aerial remote sensing in agriculture: A practical approach to area coverage and path planning for fleets of mini aerial robots. *Journal of field robotics*, **28**(5), 667–689. 11

BERGER, CYRILLE, WZOREK, MARIUSZ, KVARNSTRÖM, JONAS, CONTE, GIANPAOLO, DOHERTY, PATRICK, & ERIKSSON, ALEXANDER. 2016. Area coverage with heterogeneous uavs using scan patterns. *Pages 342–349 of: 2016 ieee international symposium on safety, security, and rescue robotics (ssrr)*. IEEE. vii, 14, 15

BYUN, SUNGWOO, SHIN, IN-KYOUNG, MOON, JUCHEOL, KANG, JIYOUNG, & CHOI, SANG-IL. 2021. Road traffic monitoring from uav images using deep learning networks. *Remote sensing*, **13**(20), 4027. 1

CHAZELLE, BERNARD MARIE. 1980. *Computational geometry and convexity*. Yale University. 16, 54, 83

CHEN, JIE, ZHA, WENZHONG, PENG, ZHIHONG, & ZHANG, JIAN. 2013. Cooperative area reconnaissance for multi-uav in dynamic environment. *Pages 1–6 of: Control conference (ascc), 2013 9th asian*. IEEE. 3

CHEN, XUECONG, CHEN, JINCHAO, DU, CHENGLIE, & XU, YONGQIANG. 2021. Region coverage path planning of multiple disconnected convex polygons based on simulated annealing algorithm. *Pages 238–242 of: 2021 ieee 4th international conference on computer and communication engineering technology (ccet)*. IEEE. 6, 84

CHRISTOU, IOANNIS T, & MEYER, ROBERT R. 1996. Optimal equi-partition of rectangular domains for parallel computation. *Journal of global optimization*, **8**(1), 15–34. 81

CHUNG, SOON-JO, PARANJAPE, ADITYA AVINASH, DAMES, PHILIP, SHEN, SHAOJIE, & KUMAR, VIJAY. 2018. A survey on aerial swarm robotics. *Ieee transactions on robotics*, **34**(4), 837–855. 2

DAILEY, DAVID, & WHITFIELD, DEBORAH. 2008. Constructing random polygons. *Pages 119–124 of: Proceedings of the 9th acm sigite conference on information technology education*. 25

DE FLORIANI, LEILA, & PUPPO, ENRICO. 1992. An on-line algorithm for constrained delaunay triangulation. *Cvgip: Graphical models and image processing*, **54**(4), 290–300. 69

DONG, WEI, LIU, SENSEN, DING, YE, SHENG, XINJUN, & ZHU, XIANGYANG. 2020. An artificially weighted spanning tree coverage algorithm for decentralized flying robots. *Ieee transactions on automation science and engineering*, **17**(4), 1689–1698. 11

FAHLSTROM, PAUL G, GLEASON, THOMAS J, & SADRAEY, MOHAMMAD H. 2022. *Introduction to uav systems*. John Wiley & Sons. 1

FENG, H-YF, & PAVLIDIS, THEODOSIOS. 1975. Decomposition of polygons into simpler components: Feature generation for syntactic pattern recognition. *Ieee transactions on computers*, **100**(6), 636–650. 81

FERETZAKIS, ALEXANDROS, & BADOGIANNIS, EFSTRATIOS. 2021. Unmanned aerial vehicles for mapping and inspecting historical constructions. *Pages 1126–1137 of: International conference on protection of historical constructions*. Springer. 1

GAO, GUAN-QIANG, & XIN, BIN. 2019. A-stc: auction-based spanning tree coverage algorithm formotion planning of cooperative robots. *Frontiers of information technology & electronic engineering*, **20**(1), 18–31. 11

GAO, SHENG, WU, JIAZHENG, & AI, JIANLIANG. 2021. Multi-uav reconnaissance task allocation for heterogeneous targets using grouping ant colony optimization algorithm. *Soft computing*, **25**(10), 7155–7167. 1

GEOS CONTRIBUTORS. 2021. *GEOS coordinate transformation software library*. *https://libgeos.org/*. 20

GEWALI, LAXMI P, & HADA, PRATIK. 2015. Constructing 2d shapes by inward denting. *Pages 708–713 of: 2015 12th international conference on information technology-new generations*. IEEE. 25

GREENE, DANIEL H. 1983. The decomposition of polygons into convex parts. *Computational geometry*, **1**, 235–259. 16, 54, 83

GUPTA, HIMANSHU, & VERMA, OM PRAKASH. 2021. Monitoring and surveillance of urban road traffic using low altitude drone images: a deep learning approach. *Multimedia tools and applications*, 1–21. 1

HADA, PRATIK SHANKAR. 2014. *Approaches for generating 2d shapes*. *https://digitalscholarship.unlv.edu/thesesdissertations/2182/*. 6, 25

HAGBERG, ARIC, SWART, PIETER, & S CHULT, DANIEL. 2008. *Exploring network structure, dynamics, and function using networkx*. Tech. rept. Los Alamos National Lab.(LANL), Los Alamos, NM (United States). 23

HAN, JINLU, XU, YAOJIN, DI, LONG, & CHEN, YANGQUAN. 2013. Low-cost multi-uav technologies for contour mapping of nuclear radiation field. *Journal of intelligent & robotic systems*, **70**(1-4), 401–410. 2

HATTENBERGER, GAUTIER, BRONZ, MURAT, & GORRAZ, MICHEL. 2014. Using the paparazzi uav system for scientific research. *Pages pp–247 of: Imav 2014, international micro air vehicle conference and competition 2014*. 3

HERRING, JOHN, *et al.* 2011. *Opengis® implementation standard for geographic information-simple feature access-part 1: Common architecture [corrigendum]*. https://www.ogc.org/standards/sfa. 21

HERT, SUSAN, & LUMELSKY, VLADIMIR. 1998. Polygon area decomposition for multiple-robot workspace division. *International journal of computational geometry & applications*, **8**(04), 437–466. vii, 6, 7, 13, 14, 15, 16, 27, 30, 32, 33, 51, 53, 54, 60, 62, 63, 78, 82

HERT, SUSAN, & RICHARDS, BRAD. 2002. Multiple-robot motion planning= parallel processing+ geometry. *Pages 195–215 of: Sensor based intelligent robots*. Springer. 14

HERTEL, STEFAN, & MEHLHORN, KURT. 1983. Fast triangulation of simple polygons. *Pages 207–218 of: International conference on fundamentals of computation theory*. Springer. 16, 54

HOOK, DANIEL W., PORTER, SIMON J., & HERZOG, CHRISTIAN. 2018. Dimensions: Building context for search and evaluation. *Frontiers in research metrics and analytics*, **3**, 23. https://www.frontiersin.org/articles/10.3389/frma.2018.00023/pdf. 4

IBRAKOV, AZAT. 2022. *hypothesis-geometry*. https://pypi.org/project/hypothesis-geometry. Accessed: 06-04-2022. 25, 84

JAYAWEERA, HERATH MPC, & HANOUN, SAMER. 2021. Uav path planning for reconnaissance and look-ahead coverage support for mobile ground vehicles. *Sensors*, **21**(13), 4595. 1

KAPOUTSIS, ATHANASIOS CH, CHATZICHRISTOFIS, SAVVAS A, & KOSMATOPOULOS, ELIAS B. 2017. Darp: divide areas algorithm for optimal multi-robot coverage path planning. *Journal of intelligent & robotic systems*, **86**(3-4), 663–680. ix, 11, 78

KEIL, J MARK. 1985. Decomposing a polygon into simpler components. *Siam journal on computing*, **14**(4), 799–817. 16, 54

KETTNER, LUTZ, MEHLHORN, KURT, PION, SYLVAIN, SCHIRRA, STEFAN, & YAP, CHEE. 2008. Classroom examples of robustness problems in geometric computations. *Computational geometry*, **40**(1), 61–78. 18

KIM, JEONGEUN, & SON, HYOUNG IL. 2020. A voronoi diagram-based workspace partition for weak cooperation of multi-robot system in orchard. *Ieee access*, **8**, 20676–20686. vii, 9, 10

KIM, JEONGEUN, JU, CHANYOUNG, & SON, HYOUNG IL. 2020. A multiplicatively weighted voronoi-based workspace partition for heterogeneous seeding robots. *Electronics*, **9**(11), 1813. 9

KOUTSOUPIAS, ELIAS, PAPADIMITRIOU, CHRISTOS H, & SIDERI, MARTHA. 1992. On the optimal bisection of a polygon. *Orsa journal on computing*, **4**(4), 435–438. 33, 84

KRONENFELD, BARRY J, STANISLAWSKI, LAWRENCE V, BUTTENFIELD, BARBARA P, & BROCKMEYER, TYLER. 2020. Simplification of polylines by segment collapse: Minimizing areal displacement while preserving area. *International journal of cartography*, **6**(1), 22–46. 85

LEBEDEV, IGOR, & IZHBOLDINA, VALERIIA. 2022. Method for inspecting high-voltage power lines using uav based on the rrt algorithm. *Pages 179–190 of: Electromechanics and robotics*. Springer. 1

LEVCOPOULOS, CHRISTOS, & LINGAS, ANDRZEJ. 1984. Bounds on the length of convex partitions of polygons. *Pages 279–295 of: International conference on foundations of software technology and theoretical computer science*. Springer. 16, 54

LIEN, JYH-MING, & AMATO, NANCY M. 2006. Approximate convex decomposition of polygons. *Computational geometry*, **35**(1-2), 100–123. 16

LLOYD, STUART. 1982. Least squares quantization in pcm. *Ieee transactions on information theory*, **28**(2), 129–137. 68

LV, ZHIHAN, CHEN, DONGLIANG, FENG, HAILIN, ZHU, HU, & LV, HAIBIN. 2021. Digital twins in unmanned aerial vehicles for rapid medical resource delivery in epidemics. *Ieee transactions on intelligent transportation systems*. 1

MACIVER, DAVID, HATFIELD-DODDS, ZAC, & CONTRIBUTORS, MANY. 2019. Hypothesis: A new approach to property-based testing. *Journal of open source software*, **4**(43), 1891. 24

MARINTSEVA, KRISTINA, YUN, GENNADIY, & VASILENKO, IGOR. 2021. Delivery of special cargoes using the unmanned aerial vehicles. *Pages 1564–1587 of: Research anthology on reliability and safety in aviation systems, spacecraft, and air transport*. IGI Global. 1

MAZA, IVAN, & OLLERO, ANIBAL. 2007. Multiple uav cooperative searching operation using polygon area decomposition and efficient coverage algorithms. *Pages 221–230 of: Distributed autonomous robotic systems 6*. Springer. 27

MAZA, IVÁN, CABALLERO, FERNANDO, CAPITÁN, JESÚS, MARTÍNEZ-DE DIOS, JOSÉ RAMIRO, & OLLERO, ANÍBAL. 2011. Experimental results in multi-uav coordination for disaster management and civil security applications. *Journal of intelligent & robotic systems*, **61**(1-4), 563–585. 3

MAZA, IVAN, MUNOZ-MORERA, JORGE, CABALLERO, FERNANDO, CASADO, ENRIQUE, PEREZ-VILLAR, VICTOR, & OLLERO, ANIBAL. 2014. Architecture and tools for the generation of flight intent from mission intent for a fleet of unmanned aerial systems. *Pages 9–19 of: Unmanned aircraft systems (icuas), 2014 international conference on*. IEEE. 4

MAZA, IVAN, OLLERO, ANIBAL, CASADO, ENRIQUE, & SCARLATTI, DAVID. 2015. Classification of multi-uav architectures. *Pages 953–975 of: Handbook of unmanned aerial vehicles*. Springer. 2

MERINO, LUIS, MARTÍNEZ-DE DIOS, JOSÉ RAMIRO, & OLLERO, ANÍBAL. 2015. Cooperative unmanned aerial systems for fire detection, monitoring, and extinguishing. *Pages 2693–2722 of: Handbook of unmanned aerial vehicles*. Springer. 3

MOITRA, DIPEN. 1991. Finding a minimal cover for binary images: An optimal parallel algorithm. *Algorithmica*, **6**(1), 624–657. 81

MUÑOZ-MORERA, JORGE, MAZA, IVAN, CABALLERO, FERNANDO, & OLLERO, ANIBAL. 2016. Architecture for the automatic generation of plans for multiple uas from a generic mission description. *Journal of intelligent & robotic systems*, **84**(1-4), 493–509. 4

NEWMAN, WILLIAM M, & SPROULL, ROBERT F. 1979. *Principles of interactive computer graphics*. McGraw-Hill, Inc. 11

NIELSEN, LASSE DAMTOFT, SUNG, INKYUNG, & NIELSEN, PETER. 2019. Convex decomposition for a coverage path planning for autonomous vehicles: Interior extension of edges. *Sensors*, **19**(19), 4165. 27

NOUROLLAH, ALI, & MOVAHEDINEJAD, MOHSEN. 2017. Use of simple polygonal chains in generating random simple polygons. *Japan journal of industrial and applied mathematics*, **34**(2), 407–428. 25

PATI, KAMALJIT, BHARWANI, ANANDI, DHANUKA, PRIYAM, MOHANTY, MANAS KUMAR, & SADHU, SANJIB. 2015. Monotone polygons using linked list. *Pages 1–7 of: 2015 international conference on advances in computer engineering and applications*. IEEE. 25

PAUL CHEW, L. 1989. Constrained delaunay triangulations. *Algorithmica*, **4**(1), 97–108. 69

PINTADO, ALFREDO, & SANTOS, MATILDE. 2020. A first approach to path planning coverage with multi-uavs. *Pages 667–677 of: International workshop on soft computing models in industrial and environmental applications*. Springer. vii, 13

POLSBY, DANIEL D, & POPPER, ROBERT D. 1991. The third criterion: Compactness as a procedural safeguard against partisan gerrymandering. *Yale l. & pol'y rev.*, **9**, 301. 27

RADOGLOU-GRAMMATIKIS, PANAGIOTIS, SARIGIANNIDIS, PANAGIOTIS, LAGKAS, THOMAS, & MOSCHOLIOS, IOANNIS. 2020. A compilation of uav applications for precision agriculture. *Computer networks*, **172**, 107148. 1

RAMER, URS. 1972. An iterative procedure for the polygonal approximation of plane curves. *Computer graphics and image processing*, **1**(3), 244–256. 84

ROYO, PABLO, PEREZ-BATLLE, MARC, CUADRADO, RAUL, & PASTOR, ENRIC. 2014. Enabling dynamic parametric scans for unmanned aircraft system remote sensing missions. *Journal of aircraft*, **51**(3), 870–882. vii, 26

SADHU, SANJIB, KUMAR, NIRAJ, & KUMAR, BHUDEV. 2013. Random polygon generation through convex layers. *Procedia technology*, **10**, 356–364. 6, 25

SCHERER, JÜRGEN, YAHYANEJAD, SAEED, HAYAT, SAMIRA, YANMAZ, EVSEN, ANDRE, TORSTEN, KHAN, ASIF, VUKADINOVIC, VLADIMIR, BETTSTETTER, CHRISTIAN, HELLWAGNER, HERMANN, & RINNER, BERNHARD. 2015. An autonomous multi-uav system for search and rescue. *Pages 33–38 of: Proceedings of the first workshop on micro aerial vehicle networks, systems, and applications for civilian use*. ACM. 3

SCHIRRA, STEFAN. 1998. *Robustness and precision issues in geometric computation*. *https://pure.mpg.de/rest/items/item_1819517/component/file_2599039/content*. 18

SCHWARTZBERG, JOSEPH E. 1965. Reapportionment, gerrymanders, and the notion of compactness. *Minn. l. rev.*, **50**, 443. 27

SHAKHATREH, HAZIM, SAWALMEH, AHMAD, AL-FUQAHA, ALA, DOU, ZUOCHAO, ALMAITA, EYAD, KHALIL, ISSA, OTHMAN, NOOR SHAMSIAH, KHREISHAH, ABDALLAH, & GUIZANI, MOHSEN. 2018. Unmanned aerial vehicles: A survey on civil applications and key research challenges. *arxiv preprint arxiv:1805.00881*. 3

SHAKHATREH, HAZIM, SAWALMEH, AHMAD H, AL-FUQAHA, ALA, DOU, ZUOCHAO, ALMAITA, EYAD, KHALIL, ISSA, OTHMAN, NOOR SHAMSIAH, KHREISHAH, ABDALLAH, & GUIZANI, MOHSEN. 2019. Unmanned aerial vehicles (uavs): A survey on civil applications and key research challenges. *Ieee access*, **7**, 48572–48634. 1

SKOROBOGATOV, GEORGY, BARRADO, CRISTINA, & SALAMÍ, ESTHER. 2020. Multiple uav systems: A survey. *Unmanned systems*, **8**(02), 149–169. vii, 1, 2, 3

SLOAN, SCOTT W. 1993. A fast algorithm for generating constrained delaunay triangulations. *Computers & structures*, **47**(3), 441–450. 69

SPITZER, CARY R, & SPITZER, CARY. 2000. *Digital avionics handbook*. CRC press. 24

SUN, BING, ZHU, DAQI, TIAN, CHEN, & LUO, CHAOMIN. 2018. Complete coverage autonomous underwater vehicles path planning based on glasius bio-inspired neural network algorithm for discrete and centralized programming. *Ieee transactions on cognitive and developmental systems*, **11**(1), 73–84. 9

TANG, YUAN, ZHOU, RUI, SUN, GUIBIN, DI, BIN, & XIONG, RONGLING. 2020. A novel cooperative path planning for multirobot persistent coverage in complex environments. *Ieee sensors journal*, **20**(8), 4485–4495. vii, 11, 12

THE CGAL PROJECT. 2022. *CGAL user and reference manual*. 5.4 edn. CGAL Editorial Board. 16, 22, 83

TOR, SHU BENG, & MIDDLEDITCH, ALAN E. 1984. Convex decomposition of simple polygons. *Acm transactions on graphics (tog)*, **3**(4), 244–265. 16, 54

TORRES-GONZÁLEZ, ARTURO, CAPITÁN, JESÚS, CUNHA, RITA, OLLERO, ANIBAL, & MADEMLIS, IOANNIS. 2017. A multidrone approach for autonomous cinematography planning. *Pages 337–349 of: Iberian robotics conference*. Springer. 4

TUNG, WEN-CHIEH, & LIU, JING-SIN. 2019. Solution of an integrated traveling salesman and coverage path planning problem by using a genetic algorithm with modified operators. *Iadis int. j. comput. sci. inf. syst*, **14**(2), 95–114. 84

WANG, LINGFENG. 2004. Issues on software testing for safety-critical real-time automation systems. *Pages 530–101 of: The 23rd digital avionics systems conference (ieee cat. no. 04ch37576)*, vol. 2. IEEE. 24

WZOREK, MARIUSZ, BERGER, CYRILLE, & DOHERTY, PATRICK. 2021. Polygon area decomposition using a compactness metric. *arxiv preprint arxiv:2110.04043*. vii, 13, 67, 76, 78

XING, SHENGWEI, WANG, RENDA, & HUANG, GANG. 2020. Area decomposition algorithm for large region maritime search. *Ieee access*, **8**, 205788–205797. vii, 14

XU, MENG, XIN, BIN, DOU, LIHUA, & GAO, GUANQIANG. 2019. A cell potential and motion pattern driven multi-robot coverage path planning algorithm. *Pages 468–483 of: International conference on bio-inspired computing: Theories and applications*. Springer. 11

YAHYANEJAD, SAEED, & RINNER, BERNHARD. 2015. A fast and mobile system for registration of low-altitude visual and thermal aerial images using multiple small-scale uavs. *Isprs journal of photogrammetry and remote sensing*, **104**, 189–202. 3

ZAHARI, NM, KARIM, MOHAMMAD ARIF ABDUL, NURHIKMAH, F, AZIZ, NURHANANI A, ZAWAWI, MH, & MOHAMAD, DAUD. 2021. Review of unmanned aerial vehicle photogrammetry for aerial mapping applications. *Pages 669–676 of: Proceedings of the international conference on civil, offshore and environmental engineering*. Springer. 1

ZHIGALOVA, ALENA VALERYEVNA. 2017. *Generation of random polygons*. *https://dspace.spbu.ru/bitstream/11701/11033/1/vkr.pdf*. 25

ZHU, CHONG, SUNDARAM, GOPALAKRISHNAN, SNOEYINK, JACK, & MITCHELL, JOSEPH SB. 1996. Generating random polygons with given vertices. *Computational geometry*, **6**(5), 277–290. 25

ZIMROZ, PAWEŁ, TRYBAŁA, PAWEŁ, WRÓBLEWSKI, ADAM, GÓRALCZYK, MATEUSZ, SZREK, JAROSŁAW, WÓJCIK, AGNIESZKA, & ZIMROZ, RADOSŁAW. 2021. Application of uav in search and rescue actions in underground mine—a specific sound detection in noisy acoustic signal. *Energies*, **14**(13), 3725. 1