ULB ECOLE POLYTECHNIQUE DE BRUXELLES SABCA

Path planning for area coverage using UAV

Author :

MAVROS Mickail

Sabca's supervisor :

Melega Marco

ULB's supervisor:

Garone Emanuele

2023

Contents

List of Figures

List of Tables

Abstract

This project focuses on evaluating algorithms for optimising area coverage, in environment with or without obstacles, using Unmanned Aerial Vehicles (UAVs), with a primary emphasis on minimising travel distance and resource consumption. The approach involves strategically placing way-points and optimising UAV paths, inspired by the "Traveling Salesman Problem" (TSP), to establish an efficient cyclic path through designated areas. Different algorithms are implemented: genetic algorithm, simple local search, simulated annealing while adding comparison with "Py-Concorde", "Fast-TSP", "TSP-Solver2" and "Python-TSP" libraries. An overview of the methodology is presented, with details of the implemented algorithms, and a comprehensive benchmark analysis. The benchmark highlights the effectiveness of two algorithms, the Concorde and Fast-TSP in enabling automated UAV path planning. The developed algorithms are versatile, capable of operating within polygon-delimited regions expressed in both Cartesian and geodetic coordinates. The project encompasses a Python-based interface and an efficient C++ program, combining the flexibility of Python with the computational efficiency of C++ to provide a comprehensive solution for UAV-based area coverage.

1 Introduction

This project is dedicated to evaluating the performance of various algorithms designed to efficiently cover designated areas using Unmanned Aerial Vehicles (UAVs), in environment with or without obstacles, with a particular focus on minimising travel distance and resource consumption. The proposed solution leverages these algorithms to strategically place way-points and optimise the UAV's path, drawing inspiration from the classic "Traveling Salesman Problem" (TSP). The primary goal is to establish a cyclic path that passes through all way-points while minimising the overall travel distance, thus achieving efficient area coverage. Our developed algorithm demonstrates the capability to operate within polygon-delimited areas, where vertices can be expressed in both Cartesian and geodetic coordinates, allowing flexibility in handling diverse geo-spatial scenarios. This project comprises a Python-based main interface and an efficient C++ program. By combining the versatility of Python and the computational efficiency of C++, this project offers a comprehensive solution for optimising UAV-based area coverage. In this report, we will provide an overview of our methodology, detail the algorithms implemented, present some additional python libraries and conduct a thorough analysis of these algorithms through a benchmark process.

2 Methodology

In this section, we outline the methodology employed in this project for path planning in the context of Unmanned Aerial Vehicle (UAV) operations.

1. Geospatial Analysis

   The first step involves analysing the input area, representing it as a polygon. If necessary, geodetic coordinates are transformed to ensure uniform spatial reference. Similarly, the geometries of any obstacles within the area are analysed and converted into a suitable format for subsequent processing.

2. Point Placement

   Points are strategically placed within the polygon, ensuring that they are outside the boundaries of any identified obstacles. These points serve as key nodes for path planning.

3. Distance Matrix Computation

   A distance matrix is computed, capturing the distances between all the designated points. Penalties are applied to distances when a line connecting two points intersects with the edges of the polygon or any obstacles. This step aims to quantify the cost of traversing different paths.

4. Optimisation Algorithm Selection

   Following the establishment of the distance matrix, one of several optimisation algorithms, as detailed in the upcoming section, is selected. This algorithm will determine the optimal path considering the defined penalties and constraints.

5. Final path and Reporting

   The selected optimisation algorithm is executed to find the best possible path within the given parameters. The resulting optimal path is then visualised and documented, and the solution is stored in the "results" directory for further analysis and reference.

This methodology provides a systematic approach to path planning for UAVs within complex geo-spatial environments, ensuring efficient and obstacle-aware route determination.

3 Algorithm

After deciding the model it was mandatory to find different optimisation method to find an efficient coverage path. To do this a "Travelling salesman problem" (TSP)[1] resolution is followed, and three different algorithm were implemented to solve it.
To ensure that an efficient path is found, a specific cost function was used to ensure that no obstacles is encountered. This cost function is based on the total distance travelled (Euclidean distance) and a penalty when an obstacle is encountered (ten time the perimeter of the entire area).

3.1 Genetic algorithm

The genetic algorithm [2,3,4] (GA) create many random solutions and perform "cross-over" and "mutation" on them while at each step only keeping the best solutions. The algorithm stop when it converges on a solution (when for a certain number of iterations, no improving solution is found).

3.1.1 Cross-over

Crossover is an operation in genetic algorithms that merges genetic information from two distinct potential solutions. It serves the purpose of efficiently exploring the solution space. The crossover process typically involves three main steps:

1. Choice of parents solutions: to apply crossover effectively, it's crucial to select two parent solutions from the current population. The selection is often based on factors such as fitness or quality, and it's not limited to only the best solution.

2. Selection of the genetic interval: a random starting and ending point within the genetic representation of the parents is chosen. This interval defines the region where genetic material exchange will occur.

3. Generation of off-springs: during this step, the selected genetic interval in one parent is replaced with a corresponding section from the other parent. The goal is to create two offspring solutions that potentially inherit beneficial traits from both parents. The specific crossover method used may determine how this exchange occurs.

[1]The Travelling Salesman Problem (TSP) is a combinatorial optimisation problem where the goal is to find the shortest possible route that visits a set of given locations (cities, coordinates) exactly once and returns to the starting location

This process helps diversify the genetic pool and can lead to the discovery of improved solutions during the optimisation process.

3.1.2 Mutation

Mutation is an operation used to allow diversification of the population by adding randomness in the optimisation process, and exploring a wider scope of solutions. It perform a modification on a randomly selected interval for all the population's individuals (initial population and cross-over off-springs) by reversing the order of a part in a solution.

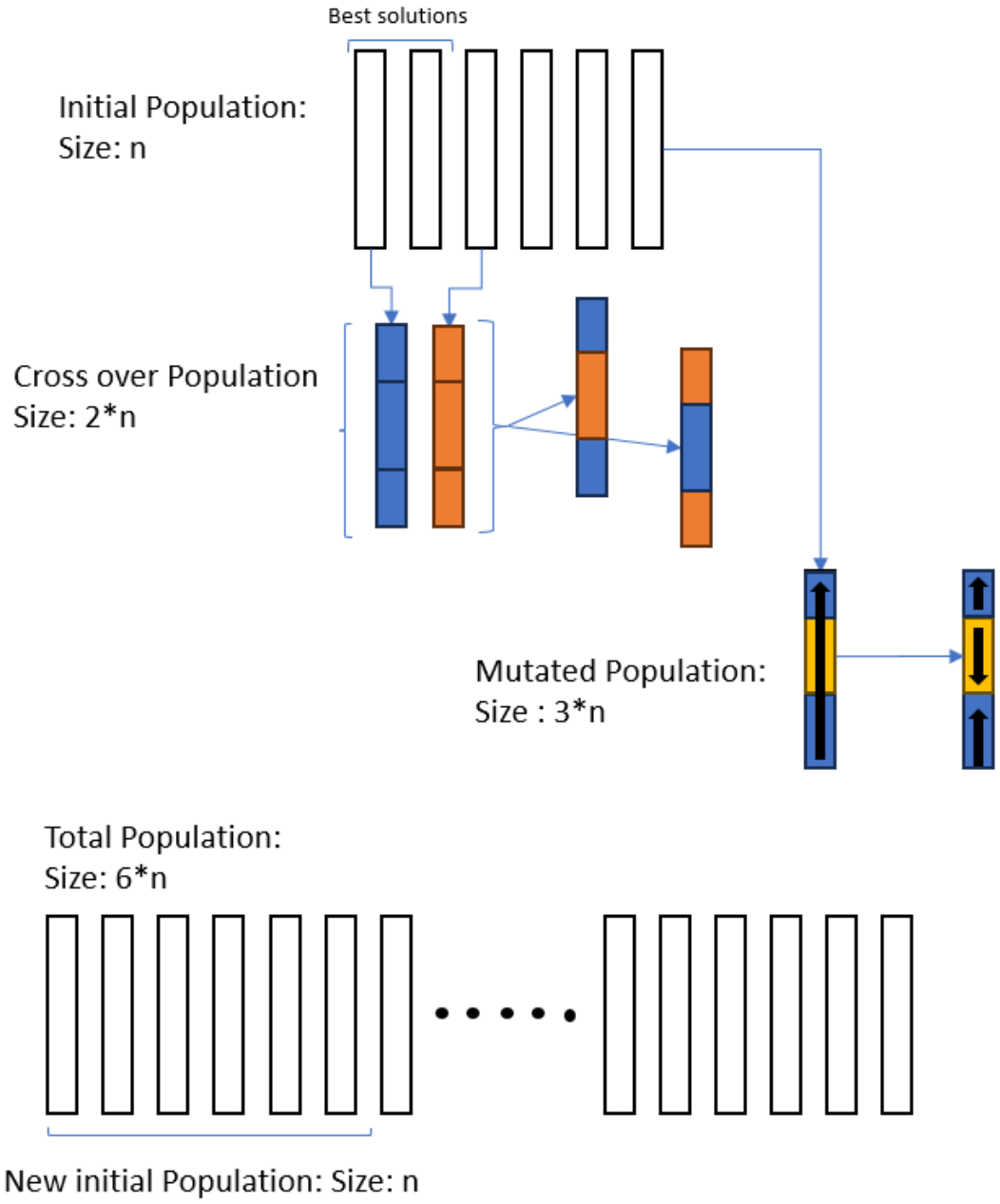

Figure 1: Different steps of a genetic algorithm

3.2 Simulated annealing

Simulated Annealing [1,4] (SA) is inspired by the annealing process in metallurgy, where materials are heated and gradually cooled to achieve a desired crystalline structure. Similarly, in optimisation, it helps "cool down" the search process from initially accepting a wide range of solutions to ultimately converging toward a high-quality solution. The Simulated Annealing algorithm begins with an initial solution and explores nearby solutions through random 2-exchange neighbourhood. Unlike a simple local search that always accepts better solutions, Simulated Annealing also has a mechanism to accept worsening solutions, using an specific acceptance criterion.

3.2.1 2-exchange Neighbourhood

In the context of Simulated Annealing, the 2-exchange neighbourhood involves swapping two elements (hence the name 2-exchange) within a solution and reversing the sequence between them. This operation is used to potentially improve the quality of the solution.

3.2.2 Temperature and cooling rate

Two crucial parameters significantly impact the performance of the Simulated Annealing algorithm: temperature and cooling rate. Those two parameter have an impact on the acceptance of worsening solution

- Temperature:
  Temperature represents the algorithm's "energy level" and influences the likelihood of accepting worsening solutions. Initially, the algorithm accepts worsening solutions with high probabilities, allowing it to explore a wide solution space. As the temperature decreases over time, the algorithm becomes more selective in accepting worsening solutions, focusing on exploitation.

- Cooling rate:
  The cooling rate determines the rate at which the temperature decreases (T = T x Cooling rate). It directly affects the balance between exploration and exploitation. A slower cooling rate allows for more exploration, while a faster rate emphasise exploitation.

This algorithm is particularly useful for escaping local optima and finding near-optimal solutions in complex optimisation problems. The choice of temperature schedule and cooling rate is critical in determining its efficiency and effectiveness.

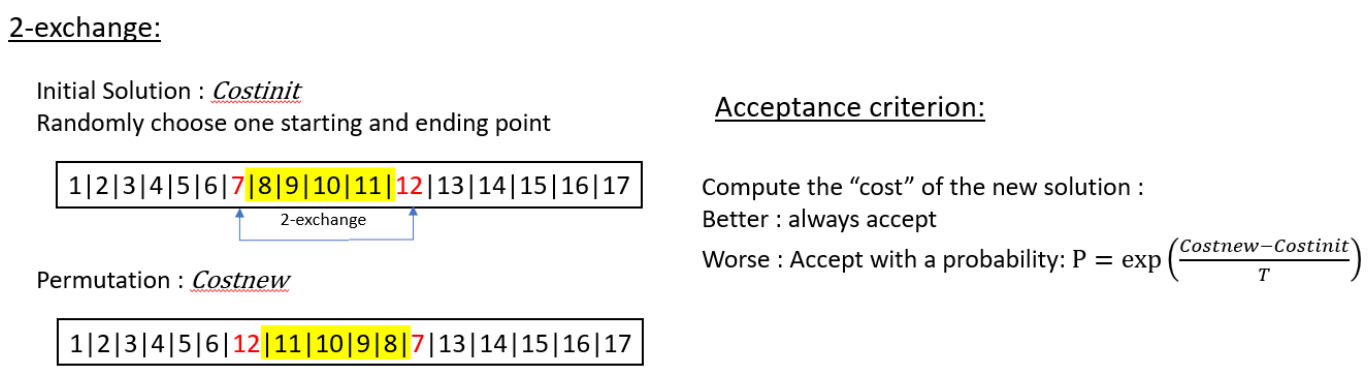

Figure 2: 2-exchange and Acceptance criterion

3.3 Simple local search

The Simple Local Search [1] (SLS) algorithm systematically evaluates all possible 2-exchange modifications in the current solution and accepts only the best improvement until no further enhancements are possible.

1. 2-Exchange Modification: The algorithm considers pairs of elements in the solution and swaps them to explore different configurations. It checks all such possible pairs to identify the one that results in the most significant improvement in the solution's quality.

2. Stopping Criterion: The algorithm continues this process until no further exchange is found that improves the solution. At this point, it has reached a local optimum, and the search terminates.

3. Deterministic behaviour: It's important to note that this method is deterministic, meaning that for a specific input, it will always return the same solution. This predictability can be an advantage or limitation depending on the problem and requirements.

4. Dependence on Initial Solution: The quality of the final solution obtained by this algorithm relies heavily on the initial solution provided. If the initial solution is close to the global optimum, the algorithm may find a good solution quickly. However, if the initial solution is far from the optimum, it may get stuck in a sub-optimal local minimum.

In summary, the Simple Local Search is a systematic optimisation approach that explores nearby solutions through 2-exchange modifications. Its final solution quality depends on the initial solution and its ability to escape local optima is limited. This algorithm can be useful for fine-tuning solutions but may not guarantee global optimality.

3.4 Theoretical comparison

These three kind of algorithms are well known and discussed in literature. Therefore, we can make an analysis of their different characteristic and expected performance.

3.4.1 Genetic Algorithm

- Advantages:
  - Global Search: Genetic algorithms are capable of exploring a large solution space efficiently, making them suitable for finding near-optimal solutions for complex TSP instances.
  - Exploration-Exploitation Trade-off: They strike a balance between exploration (searching for new solutions) and exploitation (refining the current solutions).
- Disadvantages:
  - Parameter Tuning: Genetic algorithms have several parameters like population size, crossover rate, mutation rate, etc., which require careful tuning for optimal performance.
  - Convergence Time: There's no guarantee of finding the global optimum, and convergence to a good solution can be slow, depending on the problem.
  - Computational Intensity: They can be computationally intensive, especially for large TSP instances.

3.4.2 Simulated Annealing

- Advantages:
  - Global search: Simulated Annealing can explore the solution space effectively, and it has a mechanism to escape local optima, making it suitable for TSP.
  - Parameter Sensitivity: It's less sensitive to parameter settings compared to some other algorithms like Genetic Algorithms.
  - Easy Implementation: It's relatively easy to implement and adapt to different problem instances.
- Disadvantages:
  - Parameter Tuning: While less sensitive to parameters, tuning the temperature schedule can still be challenging.

  - Convergence Time: The convergence of Simulated Annealing can be slow, especially when the temperature schedule is not well-tuned.
  - Noisy Search: It can be sensitive to the initial solution and temperature schedule.

3.4.3 Simple Local Search

- Advantages:
  - Simplicity: Simple Local Search is easy to understand and implement.
  - Quick Convergence: It can quickly converge to a local optimum, making it efficient for smaller TSP instances or as an initial solution improvement step.
- Disadvantages:
  - Local search: It is highly prone to getting stuck in local optima, making it less suitable for finding the global optimum.
  - Limited Exploration: Simple Local Search tends to focus on exploiting the current solution and lacks the global exploration capabilities of Genetic Algorithms and Simulated Annealing.
  - Noisy Search: The quality of the solution heavily depends on the starting point, and it may not find near-optimal solutions for complex TSP instances.

In summary, the choice of algorithm depends on the specific requirements of the TSP problem at hand. Genetic Algorithms and Simulated Annealing are better suited for finding near-optimal solutions in complex TSP instances, while Simple Local Search can be used for quick optimisation or as an initial solution improvement step for smaller problems. However, each algorithm has its own set of parameters and characteristics that need to be considered during implementation and tuning.

	Simple local search	Simulated annealing	Genetic
Complexity of implementation	Low	Low	High
Tuning difficulty	Zero	Medium	High
Randomness	No	Yes	Yes
Exploration of solution space	Low	Medium	High
Dependence on initial solution	High	Medium	Low
Speed	High	Medium-High	Low
Quality of solution	Medium-Low	Medium-High	High

Table 1: Theoretical algorithm comparison

4 Python libraries

This section analyse different state-of-the-arts algorithms used to solve TSP instances. A non-exhaustive list of different solver will be presented. The followed algorithms are implemented and will therefore be compared with the previously described algorithms.

4.1 Concorde

Concorde is a powerful software tool designed to solve the TSP to optimality or near-optimality for both small and large instances. It uses a branch-and-cut algorithm, which is a combination of branch-and-bound and cutting-plane methods. In this case it is implemented using the "Py_Concorde" library, which is a python wrapper around the Concorde TSP solver. Here's an overview of how Concorde works to solve the TSP:

1. Initial Solution: Concorde starts by generating an initial solution to the TSP. This solution can be obtained using a variety of heuristics or construction methods. The quality of this initial solution can impact the efficiency of the subsequent optimisation process.

2. Branch-and-Bound: Concorde uses a branch-and-bound approach to explore the solution space systematically. The goal is to divide the problem into smaller sub-problems and search for optimal or near-optimal solutions within them.

3. Branching: In the branching phase, Concorde selects a variable (e.g., an edge in the TSP graph) to branch on. It creates two child nodes—one with the variable set to "true" (indicating the presence of the edge) and the other with the variable set to "false" (indicating the absence of the edge). This branching process creates a binary tree structure.

4. Bounding: In the bounding phase, Concorde evaluates the relaxation of the TSP. This means allowing fractional values for variables (e.g., allowing partial edges in the solution) to relax the problem. The relaxation provides an upper bound on the optimal solution value. If this bound is less than the current best-known solution, the corresponding branch can be pruned, eliminating the need to explore it further.

5. Cutting Planes: One of Concorde's key strengths is its ability to incorporate cutting planes. Cutting planes are linear inequalities that can strengthen the linear programming relaxation of the TSP. These inequalities are derived from the TSP's constraints and help reduce the

relaxation's optimality gap. Concorde generates and adds these cutting planes iteratively to tighten the relaxation and improve the lower bound on the optimal solution.

6. Node Selection: Concorde employs various strategies to select nodes (subproblems) for exploration within the branch-and-bound tree. These strategies may involve selecting nodes with the largest optimality gap (the difference between the upper and lower bounds) or nodes that are most promising for finding improved solutions.

7. Pruning: Throughout the search, Concorde prunes branches that are guaranteed to yield suboptimal solutions based on the bounding and cutting plane information. This pruning helps reduce the search space and focus on promising areas.

8. Stopping Criteria: Concorde continues its search until it meets predefined stopping criteria. These criteria can include reaching a specified optimality gap, running out of computational resources, or finding an optimal solution.

9. Output: When Concorde finishes, it provides the user with the best-known solution to the TSP instance along with information about the optimality gap and other relevant details.

Concorde's combination of branch-and-bound, cutting planes, and sophisticated branching and pruning strategies makes it a powerful TSP solver capable of finding high-quality solutions for both small and large TSP instances. It has been widely used in the academic and operational communities for TSP optimisation.

4.2 TSP-Solver

TSP-Solver is a pure python program code used to find sub-optimal solutions for the TSP. It implement a simple greedy algorithm which consist in 3 main steps:

1. Input : The algorithm take as an input a matrix of distance between all the different points.

2. Initialise all vertexes, where each vertex belongs to its own path fragment. Each path fragment has length 1.

3. Find 2 nearest disconnected path fragments and connect them.

4. Repeat, until there are at least 2 path fragments.

5. Output : When the program finish it provides its final sub-optimal path and its total distance.

This greedy algorithm sometimes produces highly non-optimal solutions. To solve this, optimisation is provided. It tries to rearrange points in the paths to improve the solution (unlimited number of optimisation paths does not guarantees to find the optimal solution).

4.3 Fast-TSP

Fast-TSP is a library for computing near optimal solution to large instances of the TSP (Travelling Salesman Problem) using a local solver to allow fast optimisation process. The library is written in C++ and provides Python bindings. It implement a stochastic local search algorithm which consist of different steps:

1. Input : The algorithm take as an input a matrix of distance between all the different points, and a maximum duration time which is by default two seconds.
2. Start of with the greedy solution computed with a greed nearest neighbour algorithm.
3. Compute the cost and improve the solution using 2-opt and 3-opt neighbourhood (2-opt is similar to the 2 exchange [2] and 3-opt is a generalisation with three elements to rearrange).
4. Perform small modification to allow diversification (explore a larger search space) then perform intensification to reach local maxima.
5. Repeat diversification and intensification step until the maximum time allowed is reached.
6. Output: When the program finish it provides its best-known solution.

With all those different steps, the Fast-TSP program allow to solve large TSP instances (with thousands of points to link) and provides solutions that are of high quality. As of current development (2023), the solver has reached state-of-the-art performance.

4.4 Python-TSP

Python-TSP is a library written in pure Python for solving typical Travelling Salesperson Problems (TSP). This library implement different heuristics solver and exact solver (exact solver will not be detailed due to their low time efficiency).
The different implemented algorithm are: Simple local search and Simulated annealing.These two algorithms are already discussed and implemented (see section 3 : Algorithm).
The idea here is to visualise the differences in efficiency between the two different implementation.

5 Tests

In this section, various tests were conducted on multiple instances of the program. These tests involved the utilisation of different algorithms implemented in C++ and/or Python. Theywere applied to different types of polygons while considering varying UAV vision capabilities. The objective of these tests was to gather information regarding the algorithms' performance and stability across a range of scenarios.

The different tests focus on evaluating the algorithm's impact on the cost of a solution (the total distance of the path) and the computation time required to reach that solution.

First, some tests were executed to compare the different implemented algorithm, and then a comparison with the result obtained using all the different libraries.

5.1 Implemented algorithm

5.1.1 Polygon 1

5.1.1.1 Vision = 10

- Simple Local Search:

  Python: cost : 3534.505136053028 || Elapsed time: 865.6434 seconds

  C++: cost: 3501.4995052555723 || Elapsed time: 42.2802 seconds

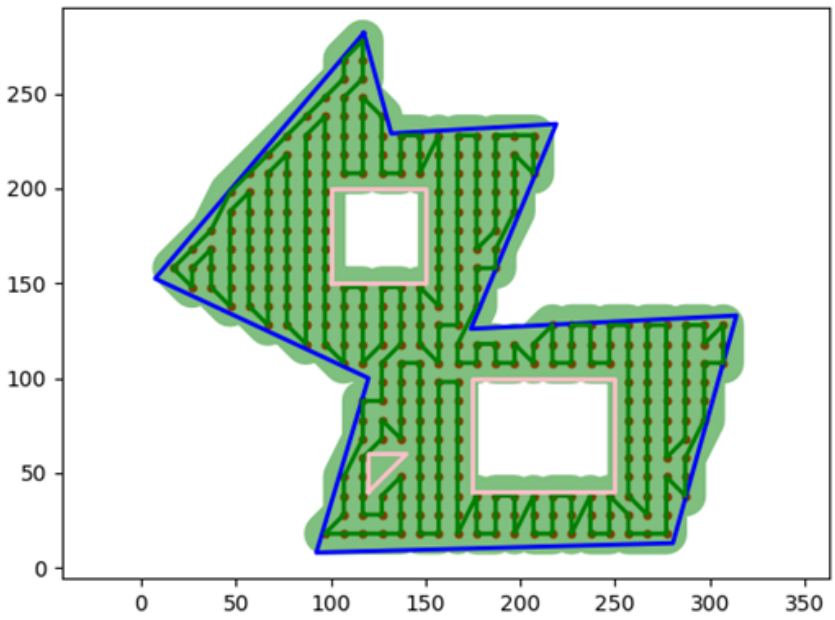

Figure 3: Nonagon with 3 obstacle with SLS resolution, vision = 10

- Simulated annealing:

  Python: cost: 3423.259018078048 || Elapsed times: 399.1184 seconds

  C++: cost: 3435.619697853045 || Elapsed times: 21.4386 seconds

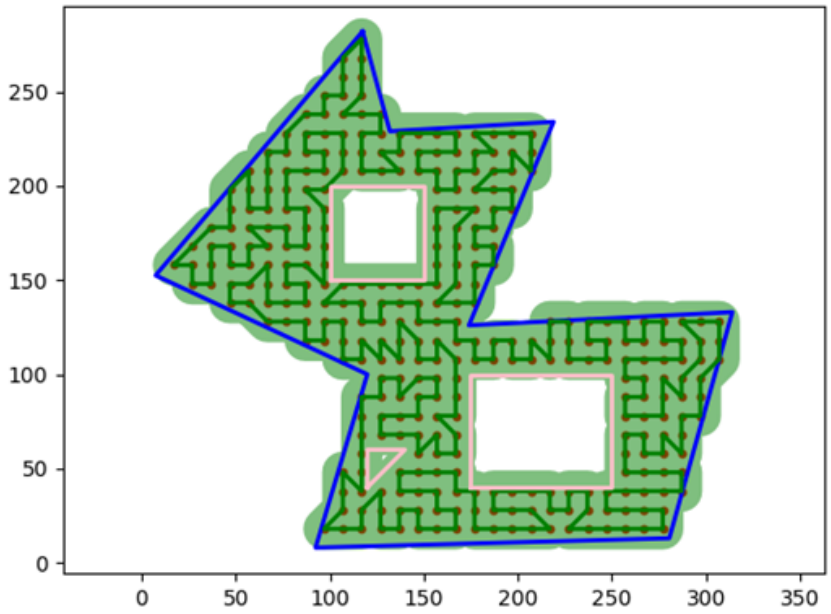

Figure 4: Nonagon with 3 obstacle with SA resolution, vision = 10

- Genetic algorithm:

  Python: cost: 3398.274750143272 || Elapsed times: 2020.3425 seconds

  C++: 3481.2489168102825 || Elapsed times: 372.2082 seconds

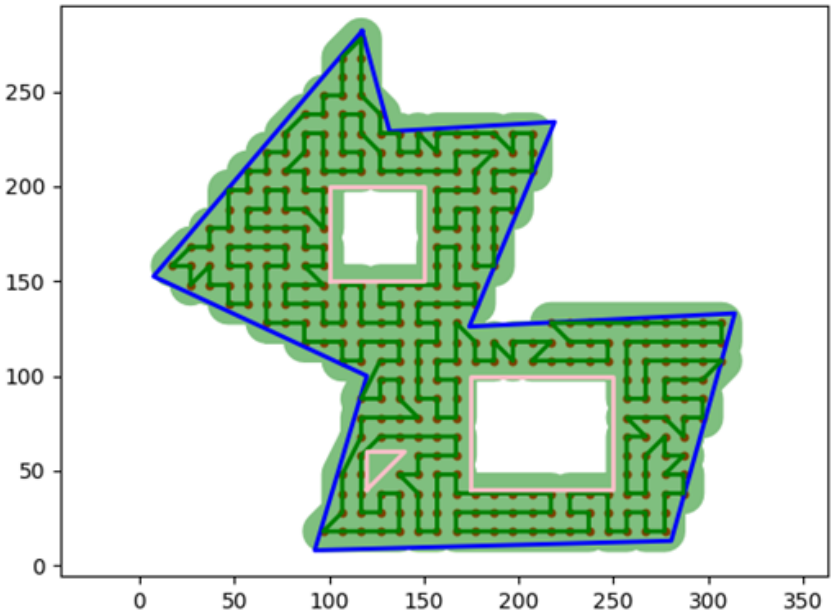

Figure 5: Nonagon with 3 obstacle with GA resolution, vision = 10

5.1.1.2 Vision = 20

- Simple Local Search:

  Python: cost: 1772.2854315746106|| Elapsed time: 3.8776 seconds

  C++: cost: 1772.2854315746108 || Elapsed time: 0.4106 seconds

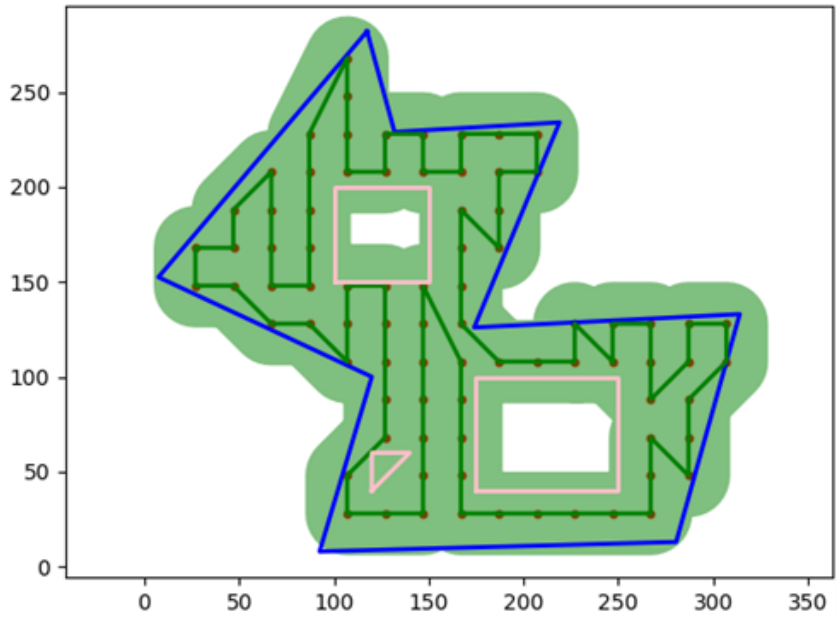

Figure 6: Nonagon with 3 obstacle with SLS resolution, vision = 20

- Simulated Annealing:

  Python: cost: 1755.7168890796868|| Elapsed times: 9.2192 seconds

  C++: cost: 1739.148346584763 || Elapsed time: 0.7162 seconds

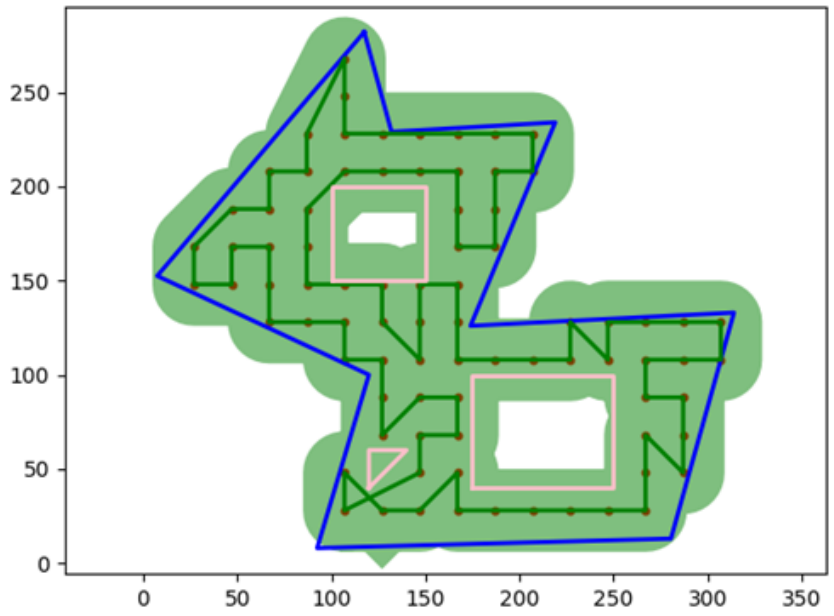

Figure 7: Nonagon with 3 obstacle with SA resolution, vision = 20

- Genetic algorithm:

Python: cost: 1890.8096252279822|| Elapsed times: 25.9792 seconds

C++: cost: 1788.8539740695346 || Elapsed time: 10.5430 seconds

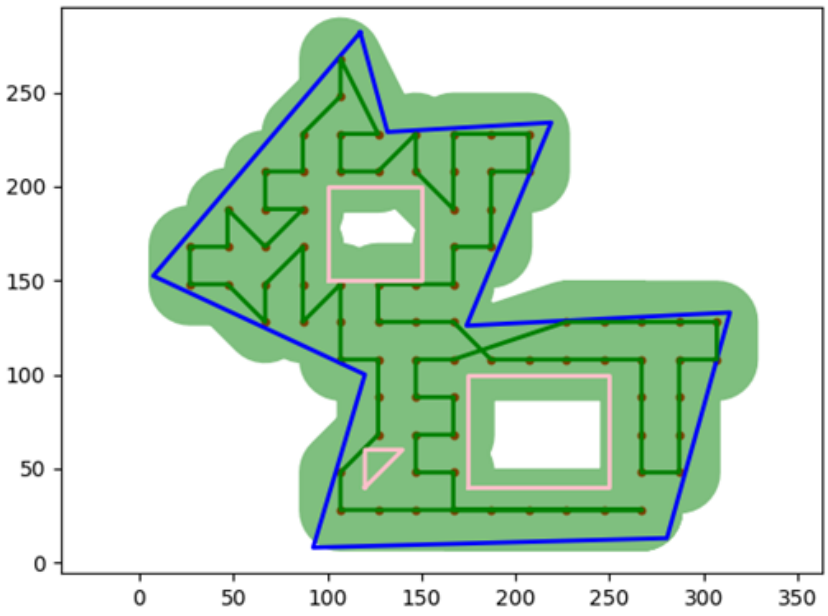

Figure 8: Nonagon with 3 obstacle with GA resolution, vision = 20

5.1.2 Polygon 2

5.1.2.1 Vision = 50

- Simple Local Search:

  Python: cost: 3042.409598129164|| Elapsed time: 0.6546 seconds

  C++: cost: 3042.409598129164 || Elapsed time: 0.0979 seconds

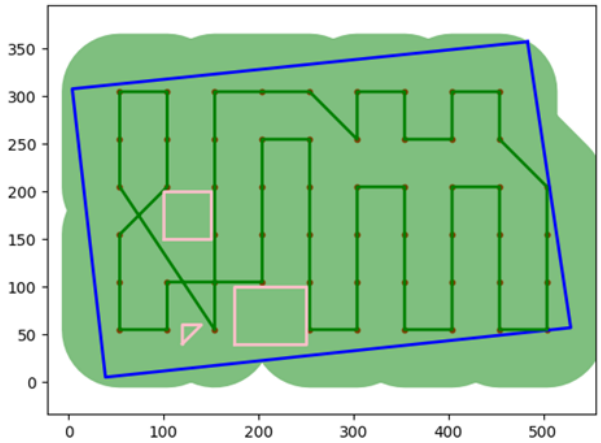

Figure 9: Rectangle with 3 obstacle with SLS resolution, vision = 50

- Simulated Annealing:

Python: cost: 2882.842712474619|| Elapsed times: 6.2527 seconds

C++: cost: 2944.6461113496084 || Elapsed time: 0.5883 seconds

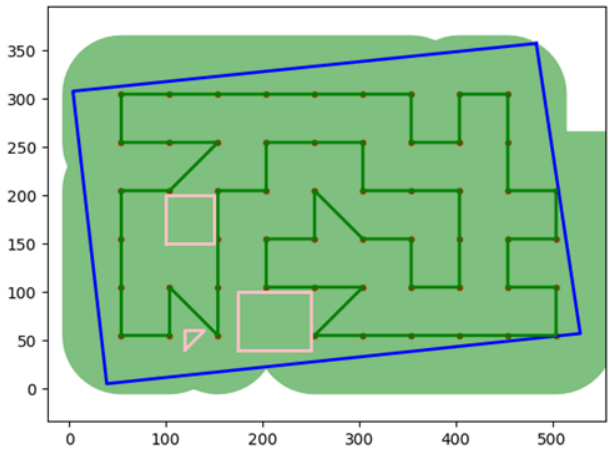

Figure 10: Rectangle with 3 obstacle with SA resolution, vision = 50

- Genetic algorithm:

  Python: cost: 2903.224755112299|| Elapsed times: 17.1796 seconds

  C++: cost: 2966.067467586918 || Elapsed time: 3.3498 seconds

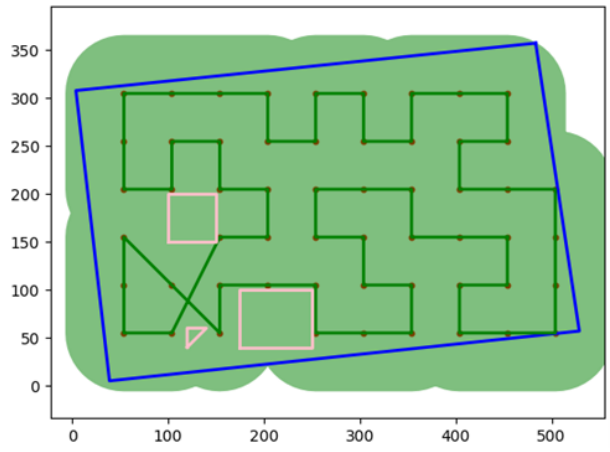

Figure 11: Rectangle with 3 obstacle with GA resolution, vision = 50

5.1.2.2 Vision = 20

- Simple Local Search:

  Python: cost: 7226.580964416991 || Elapsed time: 602.7465 seconds

  C++: cost: 7226.580964416991 || Elapsed time: 36.2240 seconds

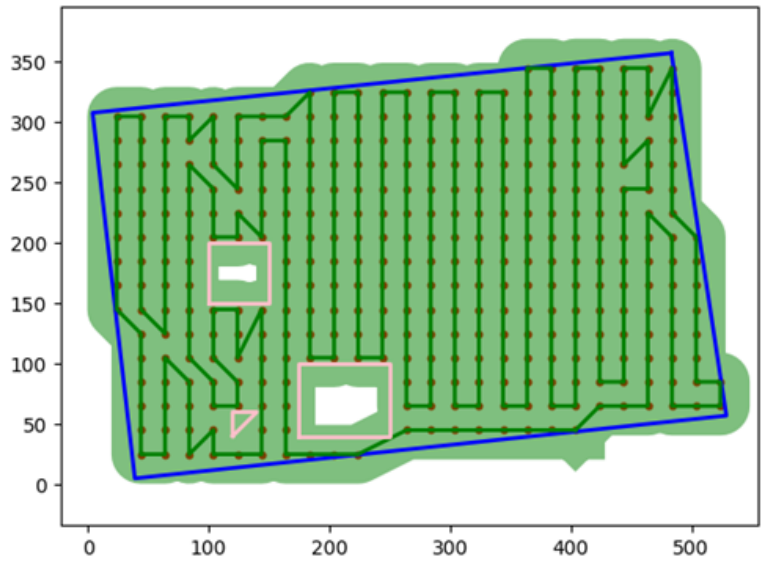

Figure 12: Rectangle with 3 obstacle with SLS resolution, vision = 20

- Simulated Annealing: Python: cost: 7218.691055746698 || Elapsed times: 395.8581 seconds

  C++: cost: 7296.680954478933 || Elapsed time: 23.0642 seconds

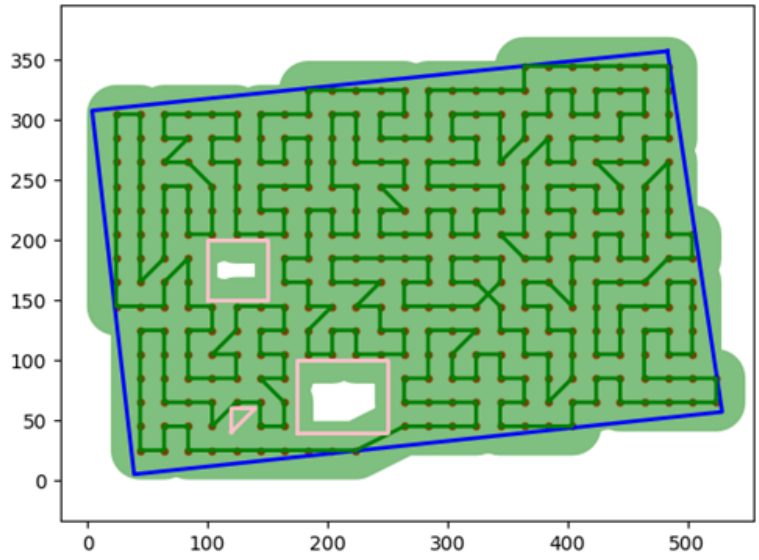

Figure 13: Rectangle with 3 obstacle with SA resolution, vision = 20

- Genetic algorithm:

  Python: cost: 7425.666382740859 || Elapsed times: 2361.0581 seconds

  C++: cost: 7656.073167240099 || Elapsed time: 408.7880 seconds

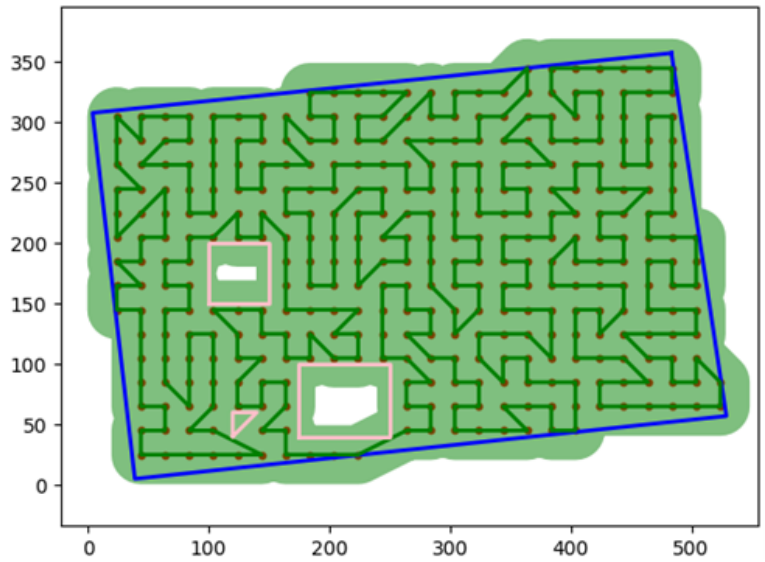

Figure 14: Rectangle with 3 obstacle with GA resolution, vision = 20

5.1.3 Polygon 3

5.1.3.1 Vision = 100

- Simple Local Search:

  Python: cost: 51079.75043234278 || Elapsed time: 1818.9912

  C++: cost: 51079.75043234278 || Elapsed time: 109.6704 seconds

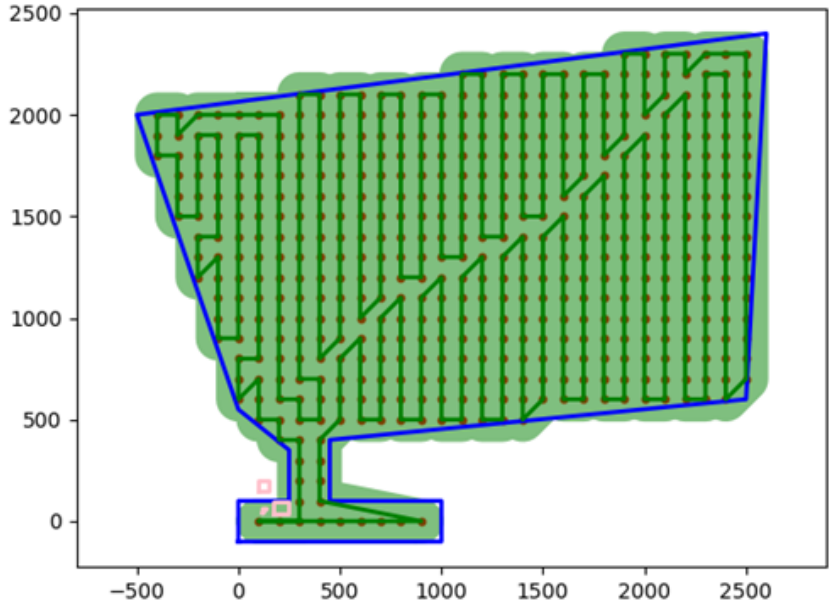

Figure 15: Dodecagon with 3 obstacle with SLS resolution, vision = 100

- Simulated Annealing: Python: cost: 51493.96399471589|| Elapsed times: 6266.2845 seconds

  C++: cost: 51328.27856976665 || Elapsed time: 298.3505 seconds

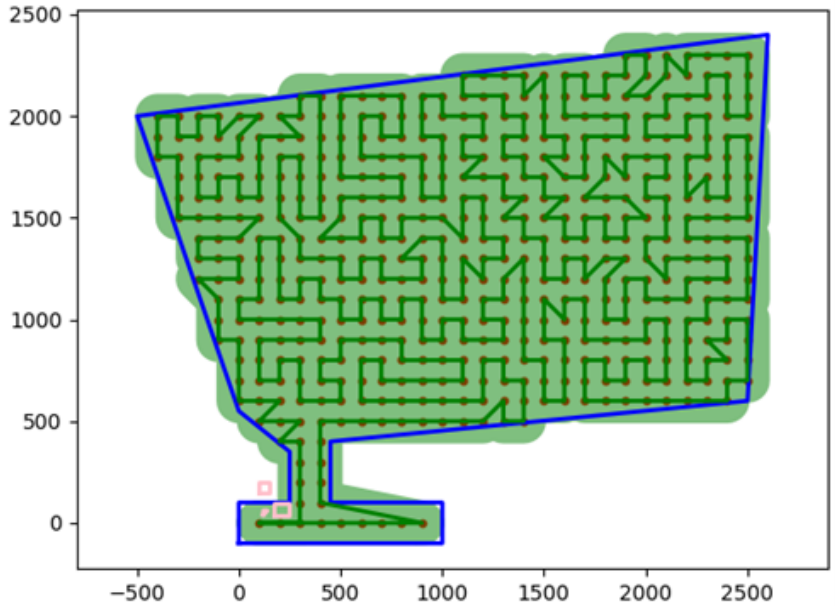

Figure 16: Dodecagon with 3 obstacle with SA resolution, vision = 100

- Genetic algorithm:

  Python: cost: 52653.761969360545 || Elapsed times: 6152.9959

  C++: cost: 53644.93427880264 || Elapsed time: 1557.8439 seconds

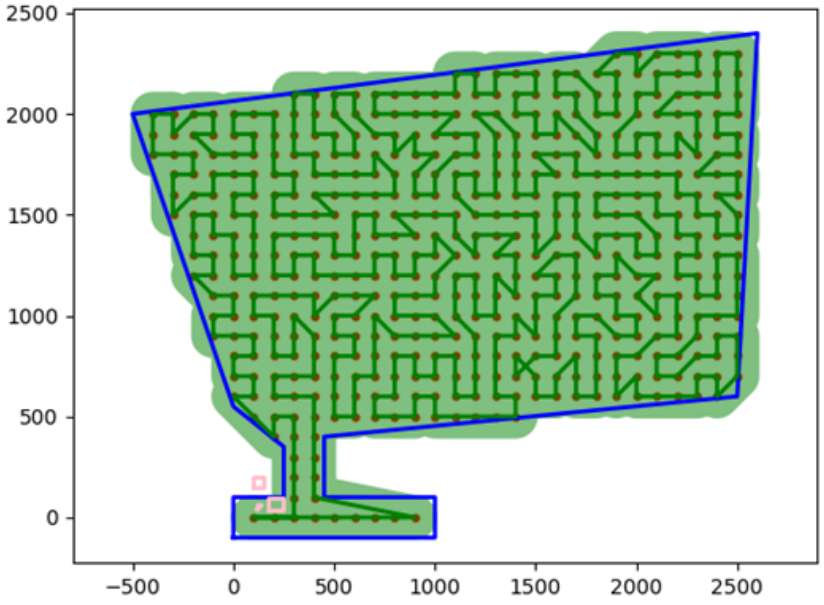

Figure 17: Dodecagon with 3 obstacle with GA resolution, vision = 100

5.1.4 Polygon4

5.1.4.1 Vision = 1

- Simple Local Search: C++: cost: 1526 || Elapsed time: 9955.5022 seconds

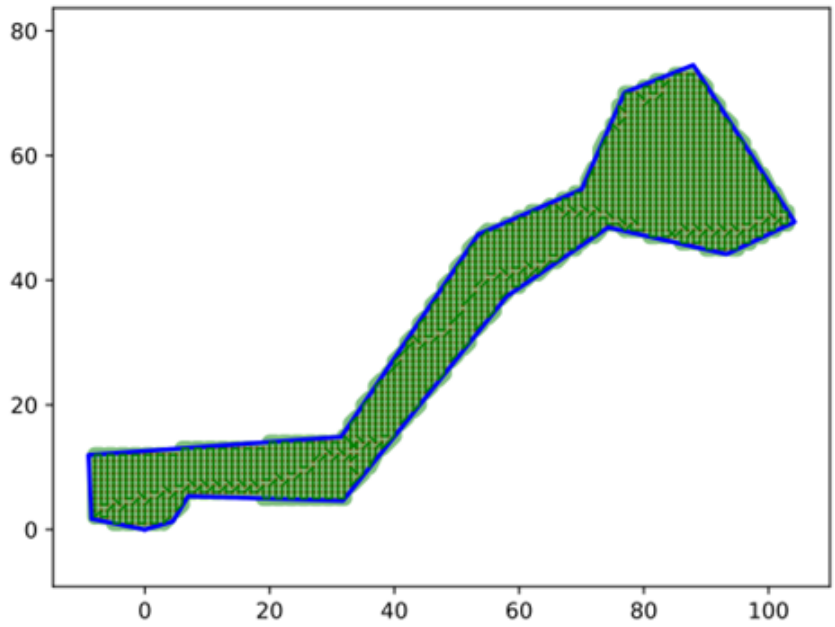

Figure 18: Pentadecagon with 3 obstacle with SLS resolution, vision = 1

- Simulated Annealing: C++: cost: 1515 || Elapsed time: 10241.0901 seconds

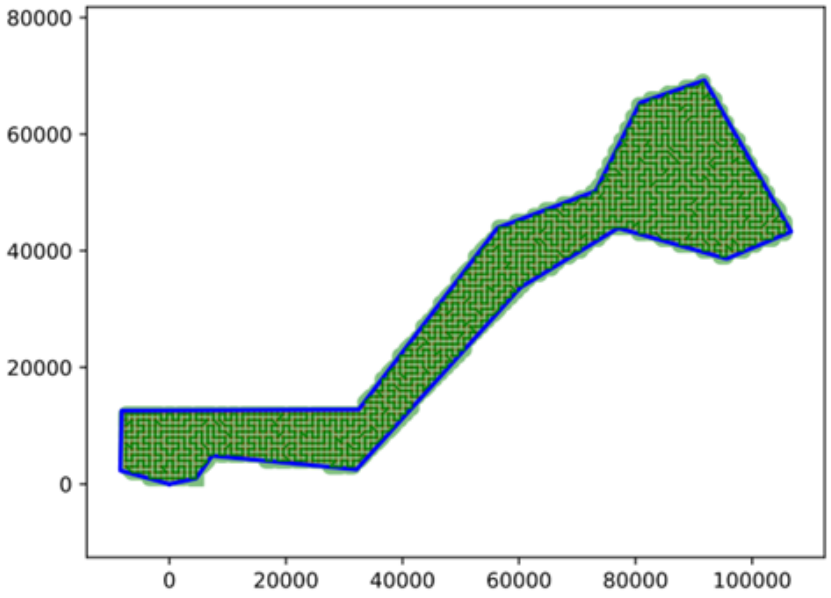

Figure 19: Pentadecagon with 3 obstacle with SA resolution, vision = 100

- Genetic: C++: cost: || Elapsed time: took too much time to reach a solution.

5.1.4.2 Vision = 2

- Simple Local Search: C++: cost: 758|| Elapsed time: 79.8338 seconds

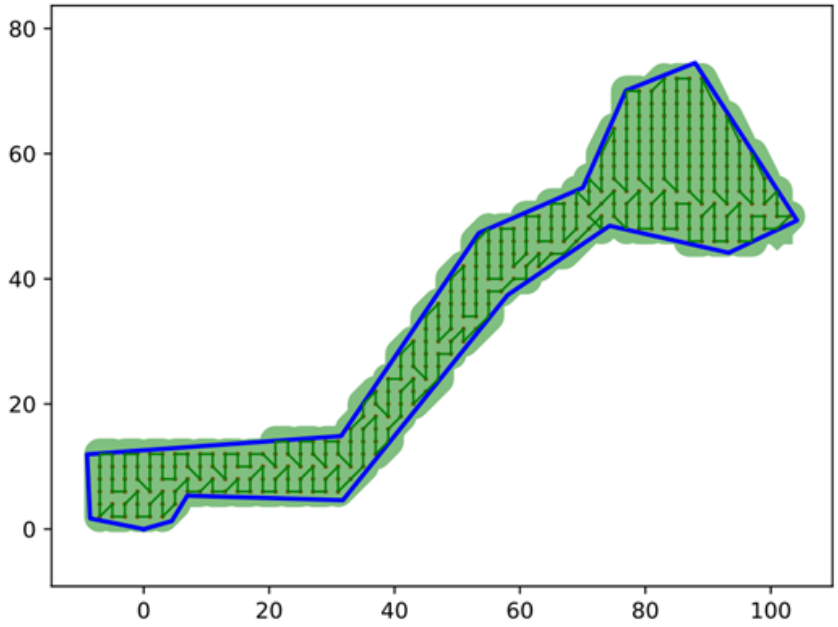

Figure 20: Pentadecagon with 3 obstacle with SLS resolution, vision = 2

- Simulated Annealing: C++: cost: 881.7199128346465 || Elapsed time: 25.8075 seconds

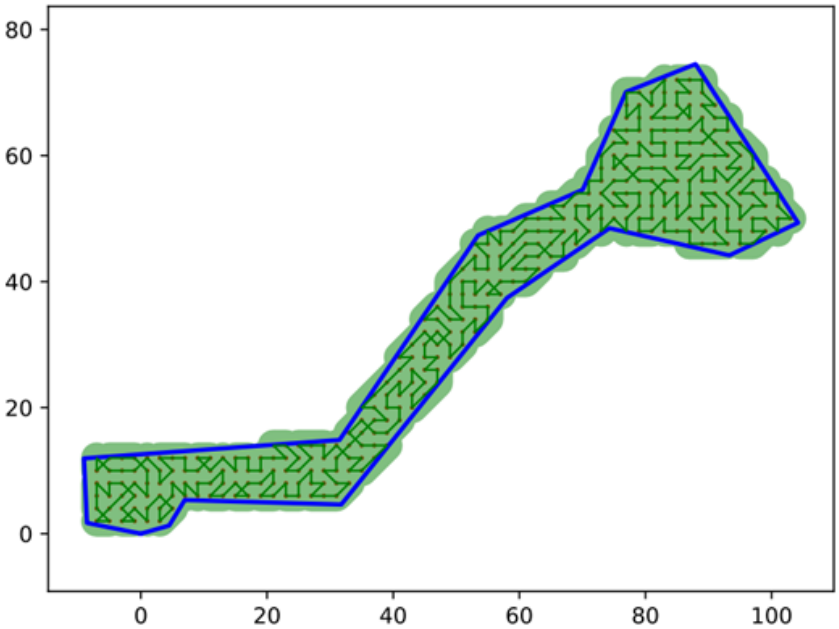

Figure 21: Pentadecagon with 3 obstacle with SA resolution, vision = 2

- Genetic algorithm: C++: cost: 915.5169984870319 || Elapsed time: 559.0512 seconds

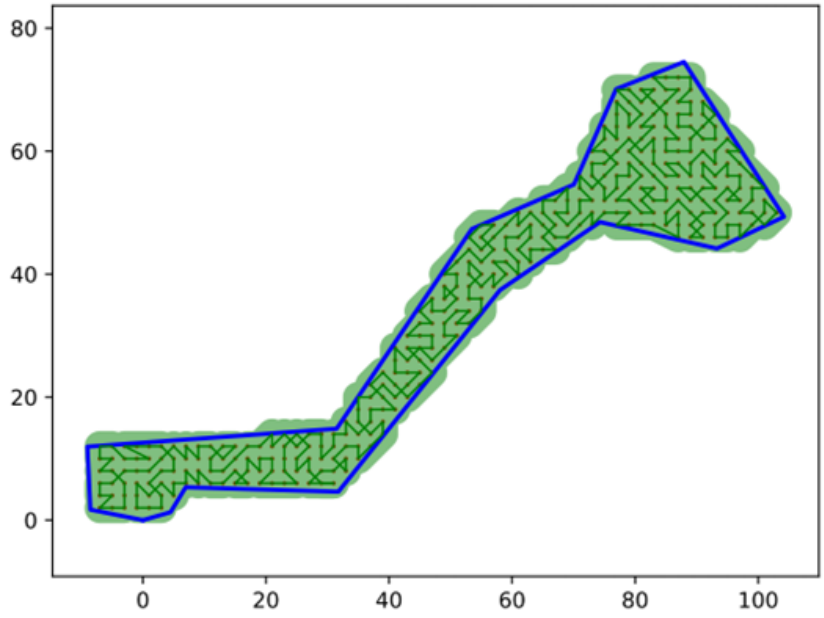

Figure 22: Pentadecagon with 3 obstacle with GA resolution, vision = 2

5.2 Global tests

The objective of this section is to compare all the different algorithm and observes their difference in term of final solution distance and computation time . All the different logs and output can be found in the results folder.[2]

	SA	GA	SLS	Concorde	Fast-TSP	Solver2	Py-SA	Py-SLS
A0 : R = 20	4493	4591	4459	4264	4264	4546	4627	4615
A0 : R = 12	7298	7660	7215	7064	7064	7644	7779	7885
A1 : R = 16	1434	1423	1509	1298	1298	1710	1452	1506
A1 : R = 10	1992	2097	2007	1950	1950	2371	2164	2209
A2 : R = 150	19591	20647	20013	19591	19591	23037	20901	21339
A2 : R = 90	34430	35506	34746	33670	33670	34986	36422	36260
A3 : R = 1550	584815	608792	590220	569534	569534	689662	617993	616415
A4 : R = 200	76092	78566	75778	73923	73927	80652	82036	79805
A5 : R = 8	843	853	832	832	832	832	902	901
A6 : R = 500	1387557	ND	1406067	1353757	1363052	1514454	1521214	1466491

Table 2: Comparison of the distance of the final path for the different algorithms.

	SA	GA	SLS	Concorde	Fast-TSP	Solver2	Py-SA	Py-SLS
A0: R=20	1.080	47.114	1.310	0.269	2.003	0.036	7.039	1.85
A0: R=12	24.666	391.697	34.804	0.741	2.022	0.306	76.938	28.602
A1: R=16	0.075	2.775	0.103	0.023	2.000	0.005	1.761	0.124
A1: R=10	0.983	18.087	1.184	0.363	2.002	0.042	3.542	1.147
A2: R=150	0.721	7.212	0.263	0.124	2.001	0.013	1.623	0.330
A2: R=90	15.274	109.644	9.167	0.692	2.008	0.170	20.629	5.879
A3: R=1550	15.546	137.892	11.484	0.316	2.007	0.115	26.265	5.654
A4: R=200	14.177	87.398	3.827	0.413	2.008	0.129	15.998	5.319
A5: R=8	0.678	4.507	0.110	0.121	2.001	0.005	1.190	0.300
A6: R=500	9047	ND	9471	3188	2.344	8.011	2566	1670

Table 3: Comparison of the computation times for the different algorithms.

Note :

- In the table, the fast-TSP algorithm consistently runs for two seconds due to a default input computation time of 2 seconds. Even after reaching the best result, the optimisation process continues until the allotted time is up. Conversely, the algorithm stops when the time limit is reached, even if there is potential for further path improvement.
- For TSP-Solver2, in many cases, it struggles to handle constraints posed by obstacles and may produce paths that traverse obstacles or extend beyond the defined area.

[2] In the tables "A" stands for Area and "R" stands for coverage radius

6 Analysis

In this section, we will first present an analysis between the implemented algorithm and then make a global analysis with all the libraries.

6.1 Implemented algorithm

Based on the results obtained from our experiments, the following key observations emerge:

1. Algorithm Speed:
   The C++ implementations of Simulated Annealing (SA) and Simple Local Search (SLS) outperform their Python counterparts significantly. SA and SLS are 10-20 times faster when implemented in C++. Additionally, the Genetic Algorithm (GA) shows an improvement of around 6 times with C++.

2. Approach and output consistency:
   While the Simulated Annealing and Genetic Algorithm are probabilistic in nature due to their randomness, the Simple Local Search consistently produces the same output for a given input.

3. Solution Quality:
   Both SA and SLS demonstrate a trend of converging to solutions of better quality in terms of total distance compared to GA, which can be due to tuning difficulties. This suggests that SA and SLS more effectively explore the optimisation landscape.

4. Path Characteristics:
   Notably, a distinction arises in the final paths generated by SA and SLS. SLS tends to produce straight vertical lines, while SA and GA tend to generate paths with more intricate patterns, possibly due to its optimisation mechanisms and sensitivity to the initial solution.

5. Convergence Challenges:
   In some cases, SA and GA fails to converge to a local optimum, potentially leading to sub-optimal solutions. To mitigate this issue, a hybrid approach can be adopted by running the SLS algorithm after SA. This hybrid strategy leverages the partial convergence achieved by SA while ensuring the discovery of a local optimum. This hybrid approach capitalises on the partial convergence of SA while ensuring the identification of a local optima.

6. Practical Considerations:

   When choosing an algorithm, practitioners should consider factors such as the desired path complexity and the trade-off between solution quality and computation time. Additionally, utilising the hybrid SA-SLS approach could provide a comprehensive optimisation strategy catering to factors like path complexity, solution quality, and computation time.

7. Limitations:

   It is important to note that our analysis is based on specific conditions, which may not generalise to all scenarios. These conditions include: having spaces large enough to ensure the possibility of placing a two-way path, simple polygon shapes, no isolated areas, and two-dimensional coordinates.

	Simple local search	Simulated annealing	Genetic
Tuning difficulty	Zero	Medium	High
Dependence on initial solution	High	Medium	Low
Speed	High	Medium-High	Low
Quality of solution	High	High	Medium
Complexity of the final path	Low	High	High

Table 4: Practical algorithm comparison

In summary, the C++ implementations of SA and SLS prove to be efficient solutions for area coverage optimisation, offering faster computation and more efficient solutions. The choice of algorithm should be guided by the specific requirements of the task at hand, need of straight line, time allowed for computation, ...

6.2 Global analysis

In this section, we will analyse the performance of the various algorithms in terms of both the final path distance and computation time efficiency, using the value presented in the Test section [2,3].

- Final Path Distance:

  Based on the results obtained from the tables, it becomes apparent that the Concorde and Fast-TSP algorithms consistently outperform the others in terms of the quality of the final path distance. Moreover, these two algorithms often converge to the same solution. This suggests that Concorde and Fast-TSP excel in finding the optimal or near-optimal solutions for the given instances.

  Following Concorde and Fast-TSP in terms of path quality are the Simulated Annealing (SA)

and Simple Local Search (SLS) algorithms, which were implemented specifically for this project (as opposed to using Python libraries). While they may not reach the same level of performance as Concorde and Fast-TSP, they provide competitive results in terms of path quality.

In contrast, TSP-Solver2 stands out with the worst quality solutions, and in certain cases, it even produces invalid solutions by crossing obstacles or going outside the defined areas.

- Computation Time Efficiency:

  In the realm of computation time efficiency, TSP-Solver2 and Concorde emerge as the top-performing algorithms. They consistently demonstrate shorter computation times across various problem instances. Concorde, in particular, is known for its efficiency in solving large instances of the Traveling Salesman Problem.

  Fast-TSP exhibits commendable scalability and computation time efficiency, making it a favourable choice when the input conditions are chosen judiciously. It shows that the efficiency of Fast-TSP depends on the specific problem parameters.

  On the other hand, the implemented Simulated Annealing (SA) and Simple Local Search (SLS) algorithms outperform their equivalent Python libraries in terms of computation time for small instances. However, they may have limitations when it comes to scalability and handling larger problem.

In conclusion, the choice of the most suitable algorithm depends on the specific requirements and constraints of the problem at hand. If achieving the highest quality path is the top priority and computation time is not a significant concern, Concorde and Fast-TSP should be the preferred choices due to their consistent ability to find optimal or near-optimal solutions.
For scenarios where computational efficiency is critical, Concorde remain top contenders, with Fast-TSP offering a versatile option depending on the problem's characteristics.
It's worth noting that the implemented Simulated Annealing (SA) and Simple Local Search (SLS) algorithms show promise in terms of computation time efficiency compared to their Python library counterparts. However, they may not be the best choice for extremely large problem instances.

7 Conclusion

In conclusion, this project offers an automatic method for UAV-based area coverage by modelling the mission as a TSP-like problem and implementing various algorithms. Our benchmark analysis highlights the efficiency of two algorithms: Concorde and Fast-TSP. These findings enable the automation of UAV path planning while minimising travel distances.

For future upgrades, several possibilities exist. One avenue is to consider a 3D paradigm that accounts for altitude differences in the area to be covered. Additionally, exploring different algorithms for various optimisation scenarios or implementing dynamic algorithms that respond to real-time UAV information, such as adjusting paths in adverse weather conditions, are potential areas for enhancement.

Lastly, the project could evolve to address multi-UAV coverage missions, where a single UAV may not have the onboard energy to cover larger areas effectively or even be adapted to fixed wing UAVs which have limitation in their rotations and speed during the coverage missions.

Bibliography

1. Stützle T., Heuristic optimisation (INFO-H413), Université Libre de Bruxelles (ULB), Bruxelles, Belgium.

2. De Smet Y., Recherche opérationnelle (INFO-H3000), Université Libre de Bruxelles (ULB), Bruxelles, Belgium.

3. Bersini H., Techniques of artificial intelligence (INFO-H410)), Université Libre de Bruxelles (ULB), Bruxelles, Belgium.

4. F M Puspita et al 2020 J. Phys.: Conf. Ser. 1480 012029

5. L. H. Nam, L. Huang, X. J. Li and J. F. Xu, "An approach for coverage path planning for UAVs," 2016 IEEE 14th International Workshop on Advanced Motion Control (AMC), Auckland, New Zealand, 2016, pp. 411-416, doi: 10.1109/AMC.2016.7496385.

6. Bouzid, Yasser, Yasmina Bestaoui, and Houria Siguerdidjane. “Quadrotor-UAV Optimal Coverage Path Planning in Cluttered Environment with a Limited Onboard Energy.” IEEE, 2017. 979–984. Web.

7. Cook W., "Concorde TSP Solver". Concorde Home, March 2015, https://www.math.uwaterloo.ca/tsp/concorde/index.html.

8. Vankerschaver J., "pyconcorde", Github, 2nd Jully 2023, https://github.com/jvkersch/pyconcorde

9. dmishin, "tsp-solver", Github, 22th July 2020, https://github.com/dmishin/tsp-solver

10. Mulvad S., "fast-tsp", Github, 15th August 2023, https://github.com/shmulvad/fast-tsp

11. luanleonardo and Goulart F, "python-tsp", Github, 9th september 2023, https://github.com/fillipe-gsm/python-tsp