Dissertation on

"Air Traffic Management using a GPU-Accelerated Genetic Algorithm"

Submitted in partial fulfilment of the requirements for the award of the degree of

Bachelor of Technology
in
Computer Science & Engineering

UE19CS390 – Capstone Project

Submitted by:

Rahul Sanjay Rampure	**PES1UG19CS370**
Vybhav K Acharya	**PES1UG19CS584**
Raghav S Tiruvallur	**PES1UG19CS362**
Shashank Navad	**PES1UG19CS601**

Under the guidance of

Dr Preethi P
Associate Professor
PES University

January - December 2022

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY
(Established under Karnataka Act No. 16 of 2013)
100 Feet Ring Road, Bengaluru – 560 085, Karnataka, India

PES UNIVERSITY

(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

FACULTY OF ENGINEERING

CERTIFICATE

This is to certify that the dissertation entitled

‘Air Traffic Management using a GPU-Accelerated Genetic Algorithm’

Is a bonafide work carried out by

Rahul Sanjay Rampure	**PES1UG19CS370**
Vybhav K Acharya	**PES1UG19CS584**
Raghav S Tiruvallur	**PES1UG19CS362**
Shashank Navad	**PES1UG19CS601**

In partial fulfilment for the completion of the Capstone Project (UE19CS390) in the Program of Study - **Bachelor of Technology in Computer Science and Engineering** under rules and regulations of **PES University, Bengaluru,** during the period January 2022 – December 2022. It is certified that all corrections/suggestions indicated for internal assessment have been incorporated in the report. The dissertation has been approved as it satisfies the academic requirements regarding project work.

Signature Dr Preethi P Associate Professor	Signature Dr Shylaja S Sharath Chairperson	Signature Dr B K Keshavan Dean of Faculty

Name of the Examiners	**Signature with Date**
1. _____	_____
2. _____	_____

External Viva

DECLARATION

We hereby declare that the Capstone Project entitled **"Air Traffic Management using a GPU-Accelerated Genetic Algorithm"** has been carried out by us under the guidance of Dr Preethi P, Associate Professor, and submitted in partial fulfilment of the requirements for the award of the degree of **Bachelor of Technology** in **Computer Science and Engineering** of **PES University, Bengaluru** during January - December 2022. The matter embodied in this report has not been submitted to any other university or institution for the award of any degree.

PES1UG19CS370 **Rahul Sanjay Rampure**

PES1UG19CS584 **Vybhav K Acharya**

PES1UG19CS362 **Raghav S Tiruvallur**

PES1UG19CS601 **Shashank Navad**

ACKNOWLEDGEMENT

We want to express our gratitude to Associate Professor Dr Preethi P, Department of Computer Science and Engineering, PES University, for her continuous guidance, assistance, and encouragement throughout the development of this Capstone Project (UE19CS390). We are grateful to the project coordinator, Prof. Mahesh H.B., for organizing, managing, and helping with the entire process. We take this opportunity to thank Dr Shylaja S Sharath, Chairperson, Department of Computer Science and Engineering, PES University, for all the knowledge and support we have received from the department. We want to thank Dr B.K. Keshavan, Dean of Faculty, PES University, for his help. We sincerely thank Dr M.R. Doreswamy, Chancellor, PES University, Professor Jawahar Doreswamy, Pro-Chancellor, PES University, and Dr Suryaprasad J, Vice-Chancellor, PES University, for providing us with various opportunities and enlightenment every step of the way. Finally, we could not have completed this project without the continual support and encouragement from our family, friends, and staff of our department.

ABSTRACT

Air traffic management is becoming highly complex with the rapid increase in commercial and cargo flights in the USA, leading to traffic congestion in the airspace, higher flight delays, and an increased airport load. To mitigate these issues, we present a load-balanced path generation system that distributes the routes across the airspace instead of concentrating them at specific regions and delays the flight optimally if required. We utilize the extensive computing capability of a GPU by developing a fine-grained parallel genetic algorithm in CUDA, allowing the system to reach a solution that balances flight delays and air Sector Density while optimizing for individual flight costs. We also perform aircraft-to-runway mapping at every airport while ensuring that the aircraft blocks a runway for a minute during departure and arrival.

We test our algorithm for all the domestic flights in the USA for a day and show that our solution generates flight paths that result in similar flight times while providing better air traffic distribution. We demonstrate a reduction in traffic hotspots by comparing the traffic heatmaps while maintaining reasonable flight delays.

We develop a web-based user interface allowing the flight dispatcher to obtain a flight plan for prospective flights with the recommended amount of flight delay. We also provide a visual simulation for the same using an interactive simulator.

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

Air traffic has become a considerable concern resulting in the overburdening of Air Traffic Controllers (ATCs). Our work will reduce the load on each ATC by generating distributed paths for flights from their respective source to destination airports while ensuring that they do not congest the airspace. To achieve this, we consider two scenarios while generating a flight path to avoid mid-air traffic; the plane takes a longer route with lesser enroute traffic or delays itself optimally and takes a shorter approach, avoiding much of the traffic. We derive such delays by drawing a balance between the amount of delay incurred and the reduction in the amount of enroute traffic.

We model the shared airspace as a traffic network by dividing it into three-dimensional interconnected convex hulls, which we term ‘airspace sectors,’ and create a graph with such sectors as the vertices. We then develop a genetic algorithm in CUDA that generates a flight path across this network. We establish a fitness function such that the algorithm finds the best approach, where the path is neither too long nor has too much enroute traffic, while also imparting some reasonable amount of delay to the flight if required. We also consider the number of runways each airport in the USA has and map each departing and arriving aircraft at an airport to unoccupied runways, allowing it to block the allocated runway for a whole minute. While a runway is blocked, it is marked as occupied and is not mapped to no other aircraft.

We then generate a path for every flight that took place within a single day and compare the overall solution to the existing scenario by evaluating three metrics:

- Flight time; The flight time or time spent in the air by the aircraft.
- Flight Delay; The time the plane gets delayed at the departure airport.
- Sector Density; The maximum number of aircraft in a sector throughout the day.
- Sector Occupancy; The average number of aircraft in a sector throughout the day

We generalize the model presented to work on any routing network for congestion control, with an optimal trade-off between node traffic and route cost. Due to the usage of a GPU-accelerated genetic algorithm, we can consider many candidate solutions, allowing it to provide better solutions quickly. We also developed a web-based user interface, allowing the flight dispatcher to input all the prospective flights and their scheduled departure times. The algorithm outputs the optimal path to be followed by each flight and the recommended flight delay if required. We have also developed a Simulator to demonstrate the same visually.

Chapter 2

PROBLEM STATEMENT

We are generating routes for multiple aircraft from their source airports to destination airports, defined as origin-destination (OD) pairs, across an air traffic network developed in the shared airspace. We divide the airspace into 'sectors,' where each sector is a three-dimensional convex hull representing a particular volume of airspace. There exist multiple 2D static, fixed points called 'Waypoints' which are named uniquely by the civil aviation authorities like the FAA and ETMS. We formulate such sectors by clustering these waypoints using a density-based clustering algorithm. Any aircraft routed across the airspace must pass through a sequence of such sectors, which allows us to measure the 'Sector Density,' which we define as the maximum number of aircraft present in a sector throughout the day. Our objective is to ensure that we distribute the overall air traffic across all such sectors and to avoid the hotspot problem wherein too many aircraft are in a particular sector at a given time. In other words, we average the Sector Density across all the sectors instead of having a select few sectors have a very high Sector Density. We break down the problem presented into three subproblems to be solved sequentially.

We first generate the sectors using the waypoint set given as input. The size of the sectors should not be too small, allowing an aircraft to spend a considerable amount of time within it. It must not be too big, lest the algorithm should detect a sector to be congested even though the aircraft within it are at a significant separation distance. These sectors are interconnected to form an air traffic network upon which we generate the flight routes.

We then develop a genetic algorithm in CUDA, whose fitness function we set after extensive experimental testing, and establish a scheme for parallelizing the genetic operators involved. We then develop a website that provides a user interface to capture the flights to be given as input to the algorithm and executes the CUDA code at the click of a button. This website displays, for each input flight, its flight time, and the delay it would experience at the departure airport and provides a sequence of coordinates for it to follow, serving as its planned route. We also develop a simulator that visually demonstrates every flight moving across the map from their respective source airport to the destination airport as time progresses. The user can run this simulator on the website.

For measuring the performance of the algorithm developed by evaluating it against a base metric, we provide, as input, all the domestic flights that took place in a single day in the USA. We measure the aerial times, flight delays, and traffic densities for all sectors from the solution provided by our model and compare it against the same, evaluated from the paths that the input flights took in reality.

Chapter 3

LITERATURE REVIEW

In this chapter, we present the current knowledge of the area and review notable findings that help shape, inform and reform our study.

3.1 A dynamic programming approach for 4D flight route optimization [1]

- Christian Kiss-Toth et al
- 2014 IEEE International Conference on Big Data (Big Data)

3.1.1 Introduction

This paper presents a solution to the Kaggle competition GE Flight Quest 2 from where we have obtained the dataset. The problem presented in the event was a flight route optimization challenge wherein contestants were provided weather forecast information for a whole day and were required to develop a flight path generation system to minimize flight costs. The contestants' task was to optimally provide a route for each flight to reduce the average cost of the paths generated.

A flight plan is any route generated for an aircraft from its source to its destination (OD pair). It is represented as a sequential list of latitude, longitude, altitude, and airspeed quadruples. These plans are generated before departure. Since there are four dimensions involved in describing a flight plan, it can be considered a sequence of 4D points.

TABLE I: Flight plan for one plane containing 5 instructions. The altitude is given in feet, the airspeed is given in knots.

ID	Ordinal	Latitude	Longitude	Altitude	AirSpeed
32441974	1	37.07	−109.73	40000	600
32441974	2	36.86	−110.35	40000	600
32441974	3	35.23	−115.03	40000	600
32441974	4	34.43	−117.14	2000	600
32441974	5	34.40	−117.23	2000	600

The cost of a flight plan is determined by multiple factors such as fuel consumed, the delay incurred, turbulent regions passed through, altitude changes, and many others. This cost was obtained by evaluating the flight plan using a simulator provided by the competition. From a mathematical point of view, the problem was the constrained minimization of a noisy and known cost function. The flight plan must also avoid certain restricted airspace areas, and any flight plan passing through such regions incurs a considerable cost that makes it invalid.

A flight plan is considered as good, or the cost of a flight plan is low if the following conditions are satisfied:

- The 2D path between the source and destination is short and considers weather condition

- The cruising altitude is optimal, with an efficient altitude profile for ascent and descent.
- Airspeed is set to a value that provides an optimal trade-off between travelling time and fuel consumption.

The complete evaluation of the dataset provided is done in Chapter 4.

3.1.2 Three-Step Process

The optimization solution proposed is broken into three main phases:

3.1.2.1 Initial Route Generation Phase

In this phase, Dijkstra's algorithm generates a naïve shortest path that avoids passing through the restricted airspace regions, which is done by considering the intermediate vertices in the graph as the vertices of the restricted zones modelled as convex hulls (Figure 3.1). This way, the restricted airspace is avoided by the initial plan and can be passed on to the next phase for further optimization.

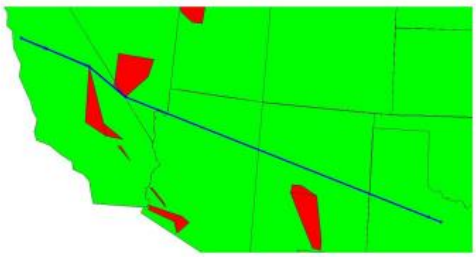


Figure 3.1

Naïve flight paths avoiding restricted zone which are shown in red

3.1.2.2 2D Optimization Phase

In this phase, the naïve path is expanded to form new routes using a dynamic programming approach wherein a grid of the airspace is formed, and alternate routes are discovered, as shown in (Figure 3.2). This method strikes a balance between minimizing route length and following wind velocity vectors, as it is more aerodynamically optimal for the aircraft heading along the wind direction (Figure 3.3).

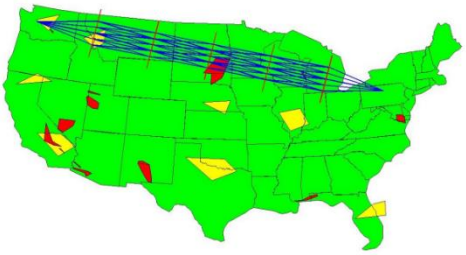


Figure 3.2

2D Optimization phase for discovering alternate routes

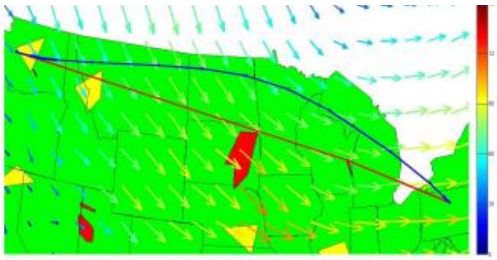


Figure 3.3

Aerodynamic efficiency of the flight paths

3.1.2.3 1D Optimization Phase

In this phase, the optimal 2D path obtained in the previous step is further optimized to take altitude into account, with an altitude profile generated that minimizes the total altitude changes or oscillations that occurred. The airspeed profile is generated, too, since optimizing the airspeed to strike a balance between delay cost

and fuel consumption would be favourable. Hence this phase is responsible for optimizing two parameters, cruise speed and altitude, which was done using an exhaustive search method.

3.1.3 Insights we can use

- Splitting the optimization process into 2D and 1D phases would modularise the program for testing different optimization techniques.
- The parameters involved in the optimization procedure include delays, airspeed, fuel, and altitude, which we need to minimize.
- Estimate the hardware requirements and problem complexity.

3.2 Priority-based Multi-Flight Path Planning with Uncertain Sector Capacities [2]

- Sudharsan Vaidhun et al.
- 2020 12th International Conference on Advanced Computational Intelligence (ICACI)

3.2.1 Introduction

The main aim of this work was to schedule multiple flights using a dynamic shortest path algorithm, using a simplified airspace model representing the airspace as a grid of sectors where uncertainties result in blocked sectors. A novel cost function based on potential energy fields is proposed for the graph search algorithm. The cost function captures the information about the blocked sectors and contending flights. A priority-based contention resolution is offered to support path planning for multiple flights in shared airspace.

3.2.2 Models and Solutions

In the model proposed, the airspace is represented by a contiguous set of sectors. The set $S = \{S_1, S_2, S_3 \ \ldots \ S_n\}$ represents all free sectors, as shown in (Figure 3.4), and each such sector is managed by its own Air Traffic Controller (ATC). Each sector has a specific sector capacity, which refers to the number of flights handled by the air traffic controller (ATC). For simplicity, we assume that each sector can manage the same number of flights and normalize the sector capacity. The sector capacity can further be reduced by adverse weather conditions for a short period. A networked Markov

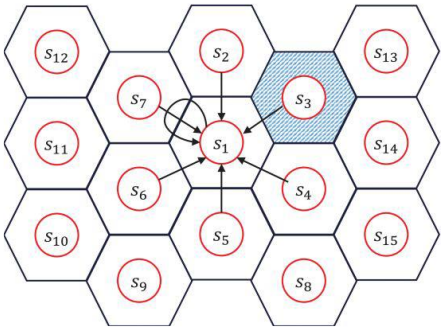


Figure 3.4

Demonstrating the sectors generated for the given airspace

process model called the Influence model can model weather impact on strategic air traffic management. The Influence Model is shown to be able to generate a realistic model of the weather dynamics. We use the influence model to model the dynamics of the sector capacity affected by the weather conditions. We consider a set of flights $F = \{F_1, F_2, F_3 \ \ldots \ F_n\}$. Each flight $F_i \epsilon F$ has a source sector S_O and destination sector S_D, where $S_O, S_D \epsilon S$. A fundamental requirement of air traffic scheduling is to allocate a path from the origin sector to the destination sector.

The solution framework proposed involved using the D* algorithm to plan the paths, and then a cost function was built based on the potential energy concept as the heuristic. The framework for working with multiple flights is based on the D* Lite algorithm, which maintains two different cost estimates for route cost from source to destination. The first estimate, '*G,'* is already known based on the information about the map, and the second estimate, '*RHS,'* is a look-ahead estimate. When the two estimates, *G* and *RHS,* are different, the node in the graph is marked as inconsistent and is placed in a priority queue. The nodes in the queue with a lower cost estimate to the destination take a higher priority in the queue. Initially, every node in the map assumes that its cost to the destination is infinite due to the map not being explored. The map exploration in a D* Lite algorithm starts from the destination until the exploration reaches our current position. The D* algorithm, therefore, does not explore the whole grid but rather fine-tunes it around the source and destination, converging to the best path in a relatively fast time. When the weather changes in a particular area, the updates are propagated to the given node and, therefore, can be handled easily.

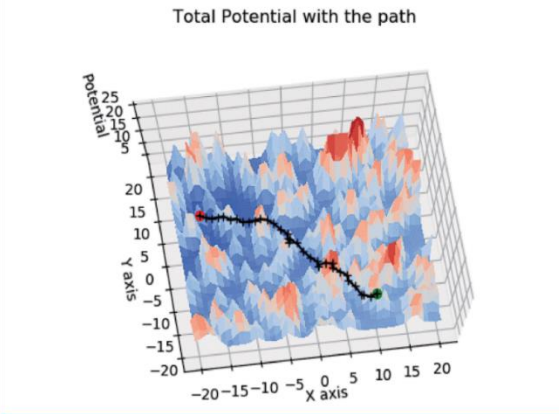


Figure 3.5

The Path generated which goes through the sectors with minimum potential

The cost function utilized a look-ahead factor based on the concept of potential energy, where the result is to always go towards the place with the least potential energy. So, we consider obstacles as high energy and the destination as 0 energy since we must hit the destination. The potential energy concept has the problem of local minima, which means that we may never reach the destination; therefore, using nodes with a distance matrix, the global minima are always considered.

With the transition cost between adjacent sectors defined, we apply the path planning algorithm for the case of a single flight in the airspace. As an example, consider the use-case scenario with the airspace of size 41×41 with 200 randomly chosen blocked sectors for a single flight. The source sector of the flight is randomly selected as the sector corresponding to $(11, -10)$ in the cartesian coordinates, and similarly, the destination sector corresponds to $(-16,5)$, as shown in

(Figure 3.5). The flight path is overlayed on the potential energy surface shown with its peaks and throughs. The track is being dynamically readjusted based on the heuristic of potential energy.

For the case of multiple flights, the approach chosen involved calculating slack time, where slack time refers to the amount of time the flight can afford to waste on its route to the destination. Based on the slack time of the flight, a priority level p is assigned; the lower value has the higher priority, and the flight with a lower slack time receives a higher priority level. The Edge cost from a sector at a higher potential to a sector at a lower potential is directly proportional to the distance between the sectors. The cost function for the path planning is designed to incur a penalty to traverse to a higher potential sector, with the penalty being proportional to the difference in potential between the sectors.

An example of this would be when certain dynamic elements change in the airspace forcing us to go to regions of High-Intensity Flights as obstacles. The blocked sectors are treated as obstacles, and a path should be made to avoid the obstacles. Similarly, under the given priority assignment, a flight with a lower priority has no control over the actions of a flight with a higher priority and therefore treats the higher priority flight as an obstacle and constructs a path to its destination. The pseudo-code for this process is shown in (Figure 3.6)

```
Input: Origin and Destination for all flights
for each flight do
    Calculate slack using Equation (7);
    Assign priority following Condition (8);
    Current sector = Origin;
    Calculate potential fields using Equations (2, 3, 4);
    Calculate the edge costs using Equation (6);
end
for each flight do
    while Current Sector ≠ Destination do
        Compute shortest path using D* Lite;
        Current Sector = Next sector;
        Update obstacles;
        Update potential fields;
        Update edge costs;
    end
end
```

Figure 3.6

The algorithm used for multi flight path planning

3.2.3 Insights we can use

- In this work, they estimated the cost based on two heuristics- static and dynamic; we could also use this idea.
- They set priority for flights based on how much ‘slack time’ each flight has, which means how much time the flight can stay without moving anywhere.
- They considered multi flights trajectories to be obstacles with high potential energy, which is something we can use.
- They used the Markov model to determine how much weather can affect the node, which we may need to do.

3.3 Application of DDPG-based Collision Avoidance Algorithm in Air Traffic Control [3]

- Han Wen et al.
- 2019 International Symposium on Computational Intelligence and Design (ISCID)

3.3.1 Introduction

The concept of dynamic rerouting in an air traffic management scenario is a complex task as it involves a centralized control that will reroute every aircraft in the system whenever anyone aircraft reroutes itself while ensuring that none of the aircraft collides with each other. In this dangerous scenario, the concept of free flight is of the essence as it becomes increasingly challenging to reroute an exceedingly high number of flights while ensuring a no-collision policy in the system. To ensure the absolute safety of free flight, it is necessary for all the flights to dynamically maintain a flight path free from all conflicts and make necessary changes to their path whenever an impending flight conflict is detected. The work presented in this paper involves using reinforcement learning for collision avoidance, focusing on the Deep Deterministic Policy Gradient algorithm, a new novel approach combining Q – Learning and Policy gradients on an Actor-Critic framework. The method offers an advantage over traditional actor-critic models since it is an "off–policy" method and relies on continuous action learning wherein the actor decides its actions directly instead of depending on probability distribution functions over its action set.

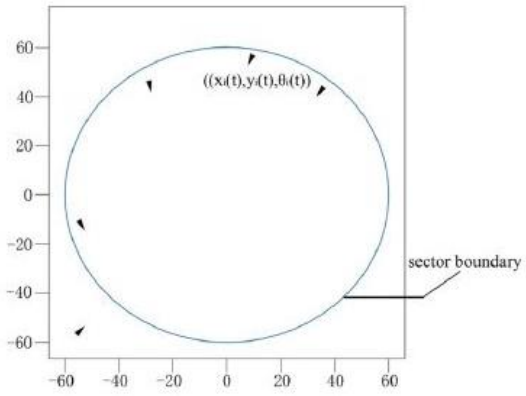


Figure 3.7

A sector containing four aircraft with one aircraft entering it

The system designed in the paper consists of a region of airspace or a sector containing a fixed number of aircraft within it, or the sector count is known at some point in time (Figure 3.7). There exists a single flight or commodity entering the sector at that time. The objective will be to achieve a system state wherein no aircraft impedes the safe radius of no other aircraft in the system, and none of the flights collides. While also ensuring that the flights can reach their desired point of exit from the sector by travelling the least possible distance. It was assumed that any disorder in the system could only be caused by the entering aircraft, which would be the only actor in the system that would alter its path trajectory. The other aircraft in the sector proceed along their current trajectory as planned without being affected by the entering aircraft.

3.3.2 DDPG Algorithm

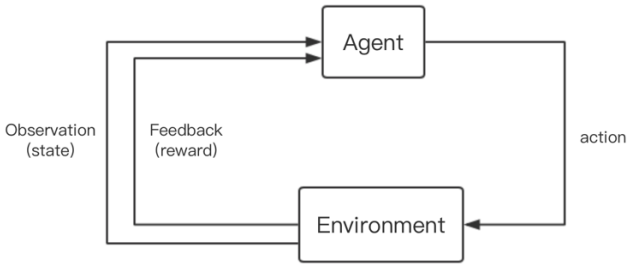


Figure 3.8

Reinforcement Learning Framework

DDPG is an algorithm that fits in the reinforcement learning domain of machine learning which is a training method based on rewarding desired behaviour and punishing undesired ones. Generally, a reinforcement learning agent can perceive and interpret its environment, take actions, and learn through trial and error. To reiterate, a model in reinforcement learning strictly refers to whether the agent is learning through environmental responses to its actions. A basic RL framework is shown in (Figure 3.8).

The DDPG algorithm uses methodologies of both DPG (Deterministic Policy Gradient) and DQN (Deep Q-Network) within an actor-critic framework set up in a multi-agent system, with two separate policies being used for exploration and updates. The original DQN worked on a discontinuous action set, with the DPG extending the DQN to a continuous action set while learning a deterministic policy.

The basic algorithm involves the usage of four neural networks:

- Actor-network:
    - The actor-network considers the entering aircraft as the actor, takes its coordinates and heading angles as the system's current state, and outputs the deflection in its heading angle as the action it takes. While in the system, these actions are taken multiple times by the actor to avoid collisions.
- Critic network:
    - The critic network takes the system state and deflection in the heading angle or the actor's action as input and returns the Q value that evaluates the action taken by the actor.
- Actor Target network and the Critic Target Network:
    - These are time-delayed copies of their original networks that slowly track the learned policies, and using these networks dramatically improves learning stability.

The two neural networks for actor and critic interchange information with each other on an alternate basis and co-learn from each other's experiences as the system progresses. The actor-

network maximizes its reward received in every iteration, eventually minimizing the overall loss function. The Bellman Equation is used to "update" the critic network to obtain a well-defined reward for the current action being taken by considering the possible future actions (which might act as a penalty). Doing so avoids the scenario where the system takes greedy actions and gives the benefit of being fast in nature due to its inherent implementation being dynamic programming/table-driven. The general system state input is shown in (Figure 3.9), and the final output is shown with the output shown in (Figure 3.10).

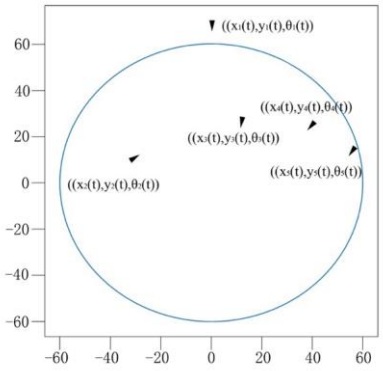


Figure 3.9

Initial System State

The general algorithm uses a minibatch to store future transitions and a relay buffer that will be a memory of past actions.

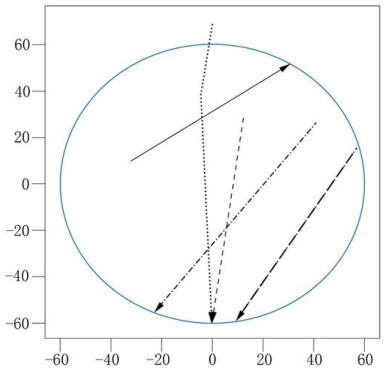


Figure 3.10

Paths followed during system progression

- At first, based on the current state of the environment and the current actor policy, the actor-network outputs a feasible action to be taken.
- Based on this action, the system's state enters a new state A, and the critic network gives the reward for this transition.
- The action taken is appended to the replay buffer (actor memory), and a new mini-batch of N transitions is generated from this state to the next state, B of the system.
- The following action is determined from this sampled policy(minibatch), and the actor policy is updated using this information.
- Using this sampled policy, the Bellman Equation is used to update the weights of the critic network to give rewards based on the new actions being taken from state B. Therefore, the critic is updated such that the loss function of the system is minimized.
- Updating the target networks is done so that the next iteration can occur.
- This process is repeated until system termination when the actor exits the sector.

3.3.3 Insights we can use

- The application of reinforcement learning for collision avoidance helps create an analogous scenario in the use-case of traffic networks, wherein the actor can be modelled as a single commodity flowing through the traffic network with the system being defined as a zone or a

sub-net within the network. The other aircraft in the various network nodes can be considered to enter and leave the system as and when they enter and leave such zones.

- The paper also gives an insight into how an autonomous system can be developed that achieves collision avoidance within an airspace sector, which we have modelled as nodes in our traffic network.

3.4 Flight Path Planning for Dynamic, Multi-Vehicle Environment [4]

- Tong He et al.
- 2020 International Conference on Unmanned Aircraft Systems (ICUAS)

3.4.1 Introduction

A novel path planning algorithm for unmanned aerial systems (UAS) flying in a dynamic environment, shared by multiple aerial vehicles, is proposed to avoid potential conflict risks. It mainly targets applications like Urban Aerial Mobility (UAM). A robust multi-staged algorithm that combines Artificial Potential Field (AFP) method and Harmonic functions with Kalman filtering and Markov Decision Process (MDP) for dynamic path planning is presented. It begins with estimating the aircraft Sector Density in the region and generates the UAS flight path to minimize collisions. The simulations of the algorithm in multiple scenarios are presented, which show adequate results.

3.4.2 The algorithm/method used

The primary algorithm suggested by this work can be broken down into three parts, each of which we summarize in the following section

- Harmonic Potential Field Formulation
- Kalman Filter
- Markov Decision Process (MDP)

3.4.2.1 Harmonic Potential Field Formulation

Harmonic Potential Field Formulation helps us to avoid the local minimum issue and is used for global path planning. We also benefit from the harmonic function here. If a function f exists whose gradient is zero, then the function f is called a harmonic function.

3.4.2.2 Kalman Filter

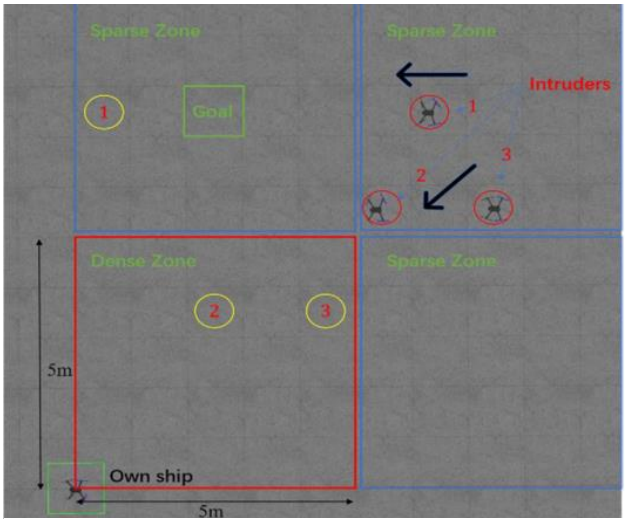


Figure 3.11

Zone creation based on Sector Density

Kalman Filter helps optimally estimate variables one needs but cannot directly access. In this paper, the Kalman Filter is used to predict the discrete states of the UAVs (Unmanned Aerial Vehicles) and facilitate dynamic path planning. This method is helpful because it helps predict the location of existing UAVs in several future instances and hence helps us segment the environment based on traffic expected at each segment. Thus, reducing the likelihood of encountering other UAVs in that area and reducing the time and effort required to execute local collision avoidance. The environment is divided into four parts, also called "zones," as shown in (Figure 3.11). The position of the intruder is tracked for the next 15 seconds. A zone can either be sparse or dense depending on the number of intruders in the next 15 seconds. A zone is said to be sparse if N equals one and is said to be dense if N is more significant than two, where N is the number of intruders expected in the next 15 seconds.

3.4.2.3 Markov Decision Process (MDP)

Markov Decision Process (MDP) is a process that predicts the future from current outcomes. MDP is used to maximize collision avoidance. MDP in this problem is defined as a tuple of 5 elements($S\ A\ P\ R\ Y$) where 'S' represents the current state, 'A' represents the action, 'P' represents state transfer probability, 'R' represents reward and 'Y' represents discount. R is the reward for going from state S to state S' by taking action A. The Kalman filter can determine the following state S.' The obstacle reward is set to a negative value, and the goal is set to a positive value. The action set A consists of directions, for example: a_{nw} would represent taking the northwest direction. If we take a specific action at a time t_k and we predict a collision at the time t_{k+1}, we use MDP to choose a different action at time t_k.To apply the MDP for obstacle avoidance, first, a local small grid map is considered around the current location, and the reward R is associated with the grid points. We can get a reward matrix, and the maximum value of the reward matrix will determine the policy $\pi(A|p)$, which will determine the direction the UAV should take.

3.4.3 Insights we can use

- This paper shows us a very robust technique to model collision avoidance effectively.

- It shows us an excellent technique to model the airspace into zones, reducing the likelihood of collisions.

3.5 A Ripple Spreading Algorithm for Free-Flight Route Optimization in Dynamical Airspace [5]

- Hang Zhou et al.
- 2020 IEEE Symposium Series on Computational Intelligence (SSCI)

3.5.1 Introduction

The work presented tries to develop a model or method for optimizing the free-flight route of one aircraft. The problem is essentially the optimization of the route trajectory. A ripple spreading algorithm is proposed to enhance existing algorithms that optimize the aircraft's free-flight route in unsteady airspace. Initially, the problem description and a mathematical model are presented wherein dynamically changing weather areas, restricted zones, and time-variant airflow characteristics are considered in the airspace. Then, a ripple spreading algorithm adapted to a dynamically weighted network is introduced. The optimal flight route can be achieved by a single run of this efficient method. The objective is to find the optimal route with minimum flight time and minimum fuel consumption since a constant TAS is assumed.

3.5.2 Algorithm/Method Used

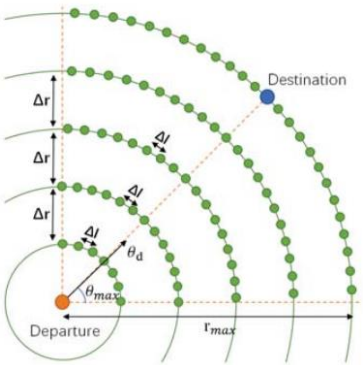


Figure 3.12
Routing Network of the graph

In the routing network, the set V consists of all nodes in the network of N nodes. The distance and bearing from (x_i, y_i) to (x_f, y_f) are denoted by r_{max} and θ_d where r_{max} represents the distance between the two points and θ_d represents the angle the vector makes with the horizontal axis. The sector was chosen: $\theta \epsilon [\theta_d - \theta_{max}, \theta_d + \theta_{max}]$ as shown in (Figure 3.12). The set L includes the links between nodes corresponding to possible free-flight routes. Direct links are added to connect the nodes to avoid the scenario where the aircraft cannot reach its destination efficiently. The weather conditions are also modelled into the airspace. Severe weather could present a serious hazard to aviation. These hazards could lead to structural damage and injury; hence, these spots are marked as "inaccessible." Optimizing a free-flight route in a dynamic environment generally uses online re-optimization (OLRO) of the routes that should be adopted. The optimal route is recalculated each time by resolving the static-path optimization based on the current environment parameters. However, the OLRO-based methods often

search for the best routes without starting from nothing but just modifying existing routes by exploring a small set of promising nodes. The problem with this approach is quite salient here. The problem is that the path given by OLRO is not necessarily optimal, but it is feasible. A method of coevolutionary path optimization (CEPO) is adopted to avoid this problem. The environmental parameters co-evolve with the route optimization process.

At an initial time, t, a ripple is generated from the source. A ripple propagates at constant speeds in all directions. When the ripple reaches an unvisited node, they are activated, and new ripples are generated from these nodes, as shown in (Figure 3.13). This process is repeated for all nodes until the destination is reached. The optimal route is generated by backtracking the steps from destination to source.

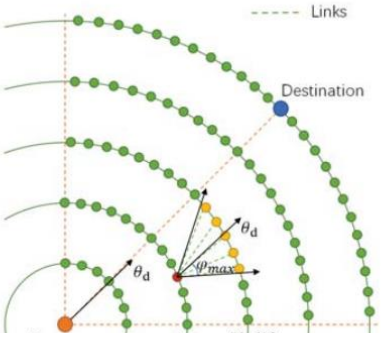


Figure 3.13

Ripple Propagation in the routing network

3.5.3 Insights we can use

- We learn potential techniques to model the weather data to our graph.
- The ripple spreading algorithm can also potentially help us in path generation.

3.6 Online free-flight path optimization based on improved genetic algorithms [6]

- Xiao-Bing Hu et al.
- Engineering Applications of Artificial Intelligence Volume 17, Issue 8, December 2004.Pages 897-907

3.6.1 Introduction

Free Flight (FF) means that the aircraft flying in shared airspace, each with its source and destinations, can do so without having to be externally coordinated and commanded by an ATC. Under FF, aircraft could fly preferred routes and have greater flexibility in manoeuvring, including “self-separation’’ from other aircraft to allow aircraft the ability to choose, in real-time, optimum routes, speeds, and altitudes, in a flexible manner and will result in the utilization of more fuel-efficient routes and a reduction in the delays imposed by communicating with the ATC. The current air traffic system is characterized by a structured airspace wherein all aircraft must fly predefined routes following the waypoints across airways. A bottleneck emerges in the system due to the entire

airspace not being utilized, resulting in a reduced air traffic capacity. A representation of aircraft movements under controlled and free flight is shown in (Figure 3.14). Potential conflicts are more difficult to predict under FF.

Optimizing fuel, time, and safety in an FF environment is a challenge that must be tackled as a complex optimization problem that is not convex and has significant non-linearities. Solving these problems in an online scenario wherein the aircraft is in motion is a complicated problem, as the priority will be on fast responses for collision resolution and safety rather than finding a function's absolute global optimal value. The work presented is a Genetic Algorithm (GA) approach that can be used for real-time flight route optimization in a dynamic environment for a single aircraft.

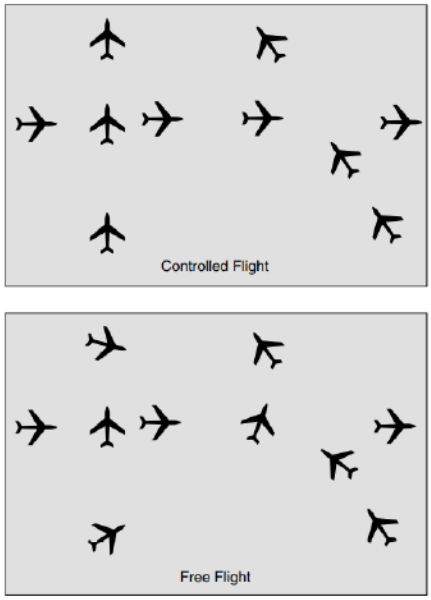


Figure 3.14
Controlled flight vs Free Flight

3.6.2 Proposed Approach and Optimizations

There are two models presented in this work, the structured approach, and the free-flight approach

Structured approach: In the traditional approach, a flight path follows a set of ground stations called waypoints. Waypoint to Waypoint navigation is carried out to determine the overall flight path. As such, waypoints/ground stations are limited, and so are the number of optional flight paths available per flight, which can be resolved by defining new waypoints in the air without any ground stations. This model is presented keeping in mind that enough waypoints were added to provide a lot of optional flight paths in the "expanded" airspace. Whenever the flight must reroute/take a different flight path, the flight will keep the current path until reaching the next waypoint and switch to the new optimal path. This method requires a central coordinator to define the new path, notify the other aircraft about this diversion, and allow them to act accordingly to adapt to the new system state.

Free-Flight approach: In the new approach, there are no waypoints, and the entire airspace is open, allowing the flight to traverse a virtually infinite number of paths from source to destination as there are no predefined routes based on waypoint constraints. In this model, the ground ATC system periodically transmits environmental and non-conflict airspace data to each aircraft. The period between successive transmissions is called a "time slice." For the aircraft, the current information is used to optimize the remaining flight path starting from the next time slice. As each time slice starts,

the new path will have a “heading” or an angle at which a straight line must be passed through with the vertical, and the aircraft will follow this straight path. The available angles at which this line can be passed through can be made discrete to some separating angle. The two models are presented in (Figure 3.15).

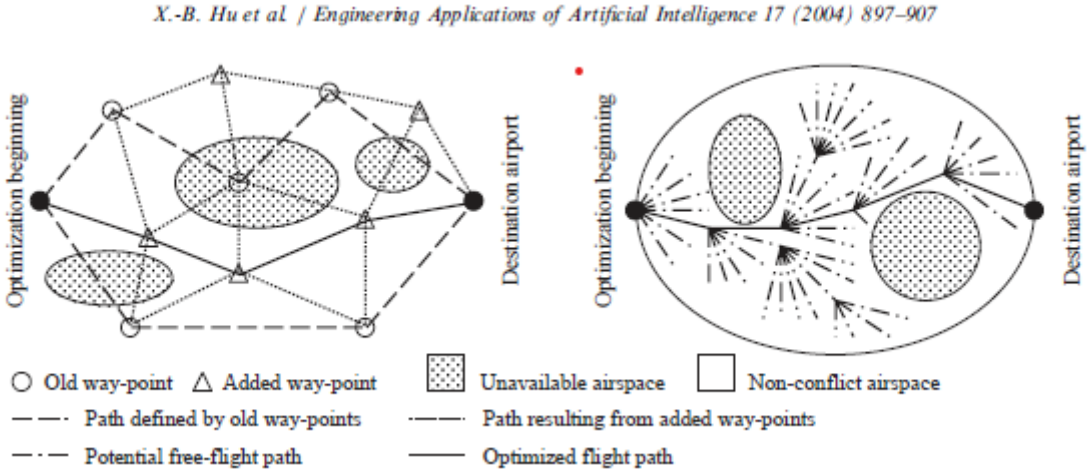


Figure 3.15

Demonstrating the two models presented

Three indices affect a flight’s cost function: fuel, time, and deviation from arrival time; this cost function acts as the fitness function, which is used for giving the optimal path starting from a waypoint. GA approach is proposed with an improvement (Figure 3.16) wherein the child chromosomes are made to go through a growing up phase during which they undergo a “survival of fittest” stage to filter out weak children - any GA optimization to speed up the process is fine. The crossover operator uses a selective (matching sections algorithm) that creates chromosomes with different paths between two intermediate waypoints.

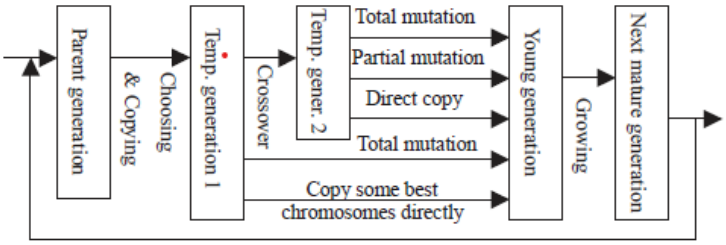


Figure 3.16

GA Framework along with the improvement method proposed

The Improved GA proposed uses a cost function composed of only one dynamic(temporal) aspect: the weather condition. Only one flight is considered, so a collision avoidance scheme is unnecessary. The cost function takes two-way points, with a line(sub-line) connected to them, and gives the cost of traversing that line. The dynamic aspect checks if adverse weather exists in the region of the sub-line at a time T and modifies the sub-line cost correspondingly. The initial population is a randomly generated set of chromosomes, each with a fixed number of genes which is the total number of waypoints available in space - each gene is a waypoint. After N generations, the fittest path (the sequence of waypoints in space) is the output. This path will have the lowest cost for

all the sub-lines along that path. This process must be repeated whenever any sub-line in the current path passes through a wrong region.

3.6.3 Insights we can use

The paper proposes a GA-based pathfinder for a single aircraft only. It does not consider a complicated cost function nor the various intricacies of air travel, like collisions with other aircraft and coordination between aircraft. Standard GA procedure is followed wherein the entire space is available for pathfinding. A long GA process scans the whole configuration space to eventually end up with the fittest solution that avoids the dynamically changing airspace conditions, which means that the set of waypoints from which paths are initialized (randomly) is extensive. Predicting if the weather becomes adverse in the future along a sub-line in the current path requires an accurate weather forecast, and such data may not be available immediately. By the time the aircraft knows it must reroute - it may not have enough time to perform the GA and reroute itself. Model 2 involves very complex mutation operators.

3.7 Co-evolving and cooperating path planner for multiple unmanned air vehicles [7]

- Chang Wen Zhenga et al.
- Engineering Applications of Artificial Intelligence Volume 17, Issue 8, December 2004, Pages 887-896

3.7.1 Introduction

UAVs must cooperate and work in a 3D dynamic environment while optimizing a cost function. Previous works do not account for these and involve heavy computation. Hence the need for coevolutionary algorithms wherein the UAVs cooperate and learn from each other's experiences is necessary. The main characteristic of this novel algorithm is that the potential paths of each UAV form their sub-population and evolve only in their sub-population. At the same time, the interactions among all sub-problems are reflected using the fitness function definition. Another essential characteristic of this approach is that individual candidates are evaluated concerning the workspace so that the computation of the configuration space (C-space) is not required. This model can generate Real-Time, desired solutions- for different kinds of mission constraints.

3.7.2 Proposed Approach

Unmanned Aerial Vehicles (UAVs) move from their source to destination following a route composed of a vector of 3D points in space. Each vehicle has its vector of 3D points, meaning that

each UAV has its route from its source to its destination. The problem boils down to generating the vector of 3D points for each UAV, such that it will either be on a point specified in its route or a line segment connecting two points on its route, always. This vector must be obtained while maintaining a minimum separation distance between any two UAVs in space. The model proposes finding the path in a workspace rather than a configuration space. The UAVs are independent of each other and evolve without the knowledge of other UAVs in space. However, the collision avoidance scheme will ensure that other UAVs' existence implicitly affects the overall solution. The model used here is like model 1 used in the previous paper, where the aircraft passes through a series of waypoints for its route and dynamically reroutes after reaching the next waypoint.

Cost Function: The cost function involves two temporal elements weather and collision avoidance technique. This cost function will judge a point in space and give an overall cost at a time T. The overall cost of any path at time T is the aggregate cost of each point in that path. The cost function includes many constraints that need to be incorporated into the overall cost of a path. The fitness function allows infeasible paths to have a valid fitness value and to participate in evolution practice. This method allows the algorithm to choose the fittest (infeasible) solution to progress the particle when feasible solutions are deemed unfit. Here the chromosome acts as a possible route a UAV can take to reach its destination. As multiple UAVs are being considered, we randomly generate the initial population, which includes N chromosomes, for each UAV. Each UAV has its population of N chromosomes. The genes are waypoints taken in the route - each path is composed of a vector of 3D points, and each 3D point acts as a gene in the path chromosome. The GA loop of selection, crossover, and mutation is executed K times by each subpopulation simultaneously.

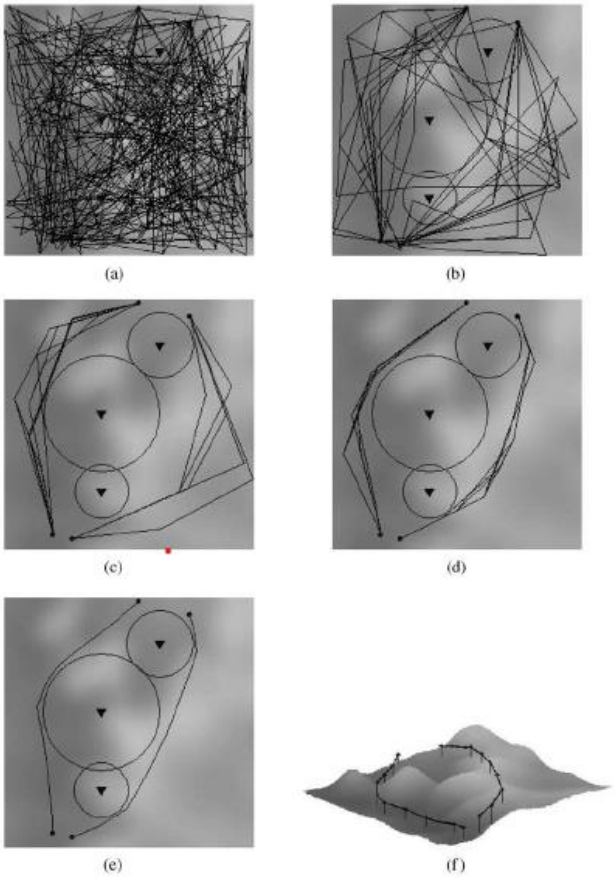


Figure 3.17

A Two UAV system with solution refinement as the system progresses

The best-fit solutions are output from each population, acting as the non-colliding optimum path for each UAV. This GA loop described above is repeated every time it is detected that an

obstacle/bad sector exists, which abruptly lowers the fitness of the current solution/path of any subpopulation/aircraft in space. As all the other subpopulations evolve, whenever dynamic rerouting is performed by one UAV, the other UAVs also are notified of such changes in the system and will reroute accordingly. Alternatively, they will maintain the same path if their previous optimal solution is still the fittest. A sample system for 2 UAVs and how the solution progresses from the initially generated random populations to the final optimal solutions is shown in (Figure 3.17).

3.7.3 Insights we can use

The work presented gave us a clear idea of how to model multiple entities in a GA framework, with each entity having its population of chromosomes that evolve based on internal and external parameters. The internal parameters enhance a single path by considering it as a single entity pathfinding problem. The solutions are enhanced based on path cost and obstacle avoidance, not considering the other entities in the system. At the same time, the external parameters enhance the solutions considering collision mitigation only; hence, the optimization is split into two independent but co-affecting phases.

3.8 Collision-free 4D path planning for multiple UAVs based on spatial refined voting mechanism and PSO approach [8]

- Yang Liu et al.
- Chinese Journal of Aeronautics Volume 32, Issue 6, June 2019, Pages 1504-1519

3.8.1 Introduction

Multi-UAV path planning is a challenging problem due to its high dimensionality, equality and inequality constraints, and the requirements of spatial-temporal cooperation of multiple UAVs, which has recently received extensive attention. The problem statement is the same as the one discussed in the previous two papers, where the goal is to find the vector of 3D coordinates for each UAV such that each UAV can follow this route given to reach its destination in the most optimum manner possible. So, the problem statement boils down to optimizing the waypoint series to achieve minimal flying time and cost.

3.8.2 Proposed Methodology

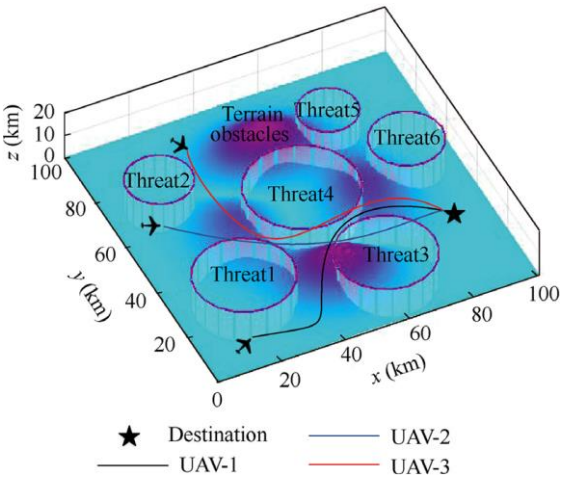


Figure 3.18

A sample system with multiple threat zones

We generate N swarms of S particles each for a list of N UAVs that need to be simulated in the system. The initialization of the position and velocity of each particle is done randomly. There should be a method that uniquely identifies each particle and gives information regarding the swarm it belongs to at the best cost it has experienced. The best cost the swarm (to which it belongs) has experienced. The spatiotemporal cost function evaluates each particle in the system to give a cost based on the particle's current position. The collision avoidance scheme gives a significant penalty to a particle if it is near a particle belonging to another swarm, thus engaging the particles of two swarms to separate from each other to avoid a collision. The paper proposes a Spatial Refined Voting Mechanism (SRVM). This PSO improvement module mitigates some of the drawbacks of standard PSO, such as converging too quickly and being unable to fine-tune its velocity to get the most optimal solution. The PSO loop is executed Gen times, and the proposed solution is obtained in the end as a series of 4D waypoints for each UAV. A sample system with an optimal solution proposed by the model is given in (Figure 3.18).

The cost function consists of two aspects: the fixed cost, which associates every point in space with a cost that remains constant, and the dynamic cost, which adds a varying amount of cost to the point depending on the time at which it was recorded. The problem is generating a vector of points where the aggregate cost at each point is minimized. We can apply the PSO technique, which finds the minima of any constrained cost function in a finite amount of time. The cost function should provide penalties (increase the cost) on a point in space if it lies within a terrain obstacle, turbulent aerial zone, or within the minimum separation sphere around another aircraft. Some penalties also depend on the aircraft's current configuration, such as a penalty for being beyond the maximum turning angle, being out of the aerial bound, or demanding the aircraft to reduce/increase its altitude by a significant factor. The paper discusses the mathematical formulae and techniques involved in calculating all such penalties and gives a final spatiotemporal cost function that optimizes using PSO. In the model presented, each particle represents a possible solution to the problem, i.e., the 4D point sequence for 1 UAV. The path is generated by following the points present at fixed distances from each other, as shown in the diagram. Dashed lines at a fixed distance from each other and Si being the ith UAV that must reach D. The path corresponds to some order of points chosen for each line with minimum overall cost. Encoding any such course into a particle is done by using a fixed-length

vector (equal to the number of lines) and initializing each vector position with a coordinate on the corresponding vertical line.

A hash map to link a path to a particle is proposed. Per the PSO loop, the velocity and position of each particle are updated by the standard PSO formula, which means that the solution/path represented by the particle must also be updated using discrete mathematical techniques. This process is done a fixed number of times, and the final solution will be the best global solution in each swarm. Whenever the current path is detected as infeasible due to the future sublines traversed by the particle passing through "bad sectors," the PSO process is repeated. The High - Level Design for the model presented in this work is given in (Figure 3.19).

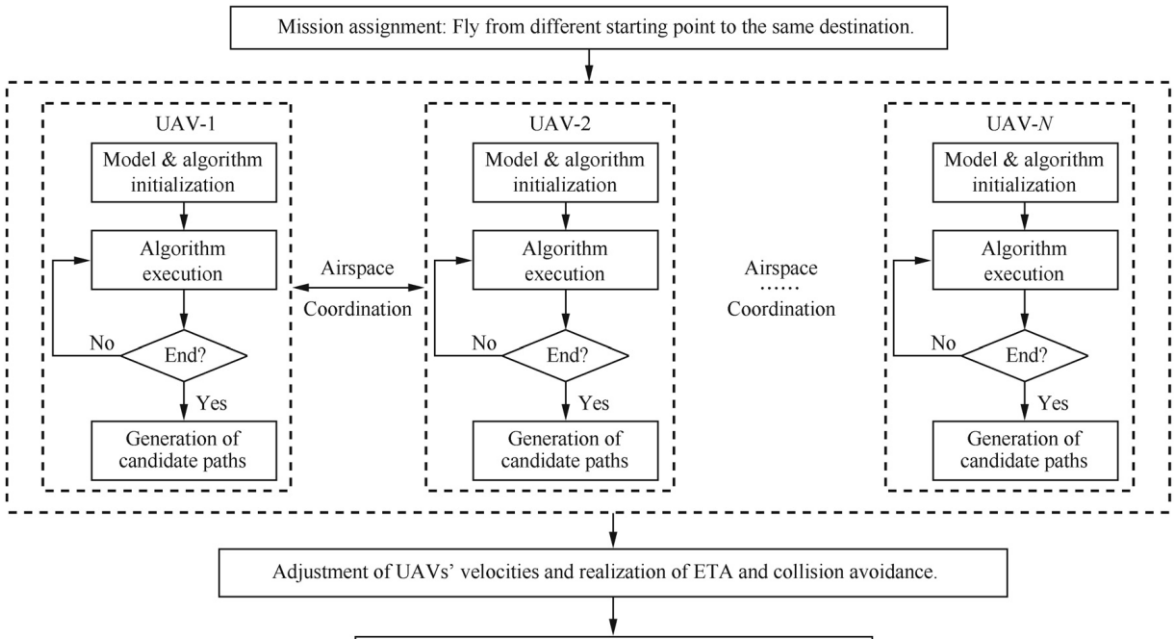


Figure 3.19

The PSO model

3.8.3 Insights we can use

- This paper estimated the cost based on two heuristics- static and dynamic; we could also use this idea.
- This paper handles all the static and dynamic costs with a fitness function and uses particle swarm optimization (PSO). We can learn how to formulate the cost functions for our given problem statement.

Chapter 4
DATASET EVALUATION

4.1 Overview

The input dataset consists of Aircraft Situation Display to Industry (ASDI) data for August and September 2013. It is taken from the Kaggle competition GE Flight Quest 2 organized by the General Electric company. The objective of the competition was to generate optimal paths for multiple test aircraft given in the dataset and evaluate the solutions in an open-source simulator provided along with the dataset. The dataset is split into three main groups of data:

- ASDI Flight History
- ASDI Flight Tracks
- ASDI Waypoints

4.1.1 ASDI Flight History

The flight history data table provides information regarding every domestic flight in the USA for August and September 2013. Around twenty thousand flights occur daily; hence about six hundred thousand flights monthly, each being associated with a row in the table. Every flight is associated with a unique Flight ID, which serves as the primary key of this table. For each flight, we have its scheduled runway departure and arrival timestamp, along with its actual runway departure and arrival timestamps. Hence, we can obtain the time it was delayed at the departure airport by its actual and scheduled runway departure timestamps. We can also get the time spent by the aircraft in the air or its aerial/flight time using the departure and arrival timestamps. Other information, such as the International Civil Aviation Organization (ICAO) codes for its departure and arrival airports, the carrier airline code, and the aircraft type, are also given.

4.1.2 ASDI Flight Tracks

The flight tracks data table consists of a sequence of coordinates for each flight in the flight history data table, representing the path taken by that flight on its journey from its source airport to its destination airport. Each coordinate is associated with the aircraft's ground speed, measured at that point, and the time when this data was recorded. We get the aircraft's cruise speed from this table by taking its mean ground speed, and we assume that the aircraft maintains this constant speed in our model for simplicity. We also plot these tracks on our sector map to get each sector's Sector Density which serves as our base metric, with which we evaluate how well our model distributes the air traffic.

4.1.3 ASDI Waypoints

Waypoints are fixed navigational aids that help flight dispatchers in providing a rough estimate of the path to be taken by the aircraft, and any route through the airspace must follow a sequence of such waypoints. For every flight in the Flight History data table, we have the path it took in the Flight Tracks data table and this corresponding sequence of waypoints in the ASDI Waypoints data table. Each row of this table corresponds to a waypoint and gives its latitude and longitude coordinates and name. By extracting all the unique waypoints from this table, we can cluster them using density-based clustering algorithms to create the airspace sectors upon which we run our algorithm.

Chapter 5

SYSTEM REQUIREMENTS SPECIFICATION

5.1 Current System

In the current system used by the aviation industry, a team of experts called flight dispatchers are stationed at an airport who use supercomputers to execute advanced flight simulation techniques to provide an optimal path for an aircraft to follow from its source to its destination airport. They generate the most aerodynamically efficient route for the plane while also considering aircraft performance and loading, forecasts of thunderstorms and turbulence, airport conditions, and airspace restrictions. Any flight scheduled to depart from an airport must first be given a route to follow by the dispatcher, which serves as its flight plan. The pilot requests this flight plan before departure and feeds the flight plan to the cockpit navigation system. On the other hand, air traffic controllers are responsible for handling airport traffic and distributing traffic in the airspace by communicating with the pilots directly via radio and radar. They notify the pilots of changes in weather forecasts, emergencies, and traffic congestion and reroute the aircraft if necessary. Based on their expertise, they essentially guide the flights to their destinations.

5.2 Design Goals

- To formulate a solution that combines the task of the ATC and the dispatcher by providing flight plans that take care of air traffic distribution while optimizing for flight cost, delays, and airport traffic.
- To ensure that the solution generates a solution in the least amount of time possible to allow for deployment in mission-critical production systems.
- To provide a user interface allowing pilots to enter their desired source and destination airport and obtain an optimal flight plan.
- To evaluate the performance of the proposed algorithm in terms of route cost, delays, and Sector Density.

5.3 Design Constraints, Assumptions & Dependencies

- We assume that provided the set of sectors to go through for each flight, the ATC, and the pilot handle in-sector navigation, which includes the usage of a collision avoidance system, autopilot system, and the take-off and landing process.

- We assume that the cruise ground speed of any aircraft remains constant throughout its journey. For simplicity, we do not consider the acceleration and deceleration that the aircraft undergoes while take-off and landing, respectively.
- We currently do not plan to factor in weather conditions or aerodynamic efficiency while generating paths and primarily plan to focus on optimizing path length, traffic distribution, and delays.
- We intend to develop the solution in CUDA, a proprietary language that allows us to create applications to run on the GPU, specifically Nvidia GPUs; hence we are dependent on using an Nvidia GPU.
- We assume that no emergencies or unpredictable scenarios occur since we do not plan to account for the same in our model.

5.4 Risks

We plan to break our model into many functional components, and the challenge would be to integrate these components into a single module. We plan to use a high-performance GPU for this problem, such as the Nvidia A100, and procuring it would be a significant issue. Building a simulator would be arduous due to the complexity involved in visualizing three-dimensional space.

5.5 Design Details

The various details involved in describing our design are split into six parts. Novelty represents what is unique in our proposed approach to solving the problem and how we differ from the work of other researchers. Innovativeness describes what improvements we offer. Interoperability delineates who will use our system and how. Performance predicts the performance of the system in terms of various metrics. Scalability represents how our model can be scaled up to real-world scenarios. Finally, resource utilization indicates the resources utilized and how this will vary with the scaling of the system.

5.5.1 Novelty

We are modelling the airspace as an air traffic network composed of a graph of convex hulls, which we obtain by clustering waypoints. Recent literature has not explored this method for representing shared airspace. We then develop a parallel genetic algorithm on CUDA with a novel fitness function and unique genetic operators, which we have not observed in contemporary academic works. We use mathematical techniques to draw the balance between air traffic decongestion and flight delays, another factor in our novelty.

5.5.2 Innovativeness

We demonstrate a solution to the air traffic management problem, where the algorithm can generate an optimal path for a flight to follow while optimizing for route cost and making sure that the enroute traffic encountered is minimized. Such a concept is innovative as we combine a flight dispatcher's core workload with an air traffic controller by ensuring traffic distribution in the flight planning stage.

5.5.3 Interoperability

We develop a website that serves as a user interface allowing a flight dispatcher to generate a flight plan quickly for any prospective flight, which is optimal in terms of cost, path traffic, and flight delays. The user will also have the option to upload a file with all the prospective flights and get a solution.

5.5.4 Performance

We develop a fine-grained parallel application in CUDA and C to maximize the performance. By running the code on the GPU, we can use a large population size, thus allowing for more solution space exploration, providing a higher chance of reaching the global optima, and leading to an overall better solution.

5.5.5 Scalability

We demonstrate that the algorithm scales up well by ensuring that the allocated memory is reused, allowing for a marginal increase in GPU memory usage as the number of input flights increases. Since the genetic algorithm is parallel, each flight takes a particular amount of time. Hence, we show that the execution time increases linearly with the input size.

5.5.6 Resource utilization

We plan to utilize an Nvidia A100 GPU via the Google Cloud Platform (GCP) since we desire to obtain the solution quickly, allowing us to perform metric analysis and improve the product if possible. We do not require much computing power in terms of CPU as we plan on only using the CPU for input-output purposes.

Chapter 6

SYSTEM DESIGN

6.1 High-Level Design

We detail the various modules involved in the application below, as well as the architecture of the website. Our dataset consists of three tables, represented in chapter 4, which we illustrate using the upright cylinder. Our website's front end runs on port 3000, and our back end runs on port 5000. We depict the various modules using rectangles and the intermediate input and output files using parallelograms (Figure 6.1).

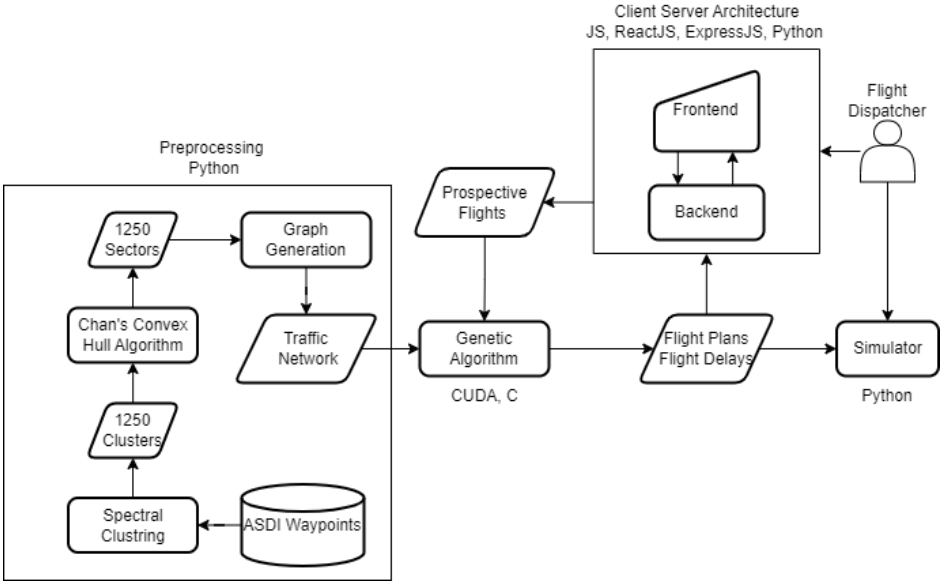


Figure 6.1 High Level Design

We also detail the design methodology for evaluating the model in Figure 6.2, where we compare our model's result with the real data and quantitatively measure the benefit provided.

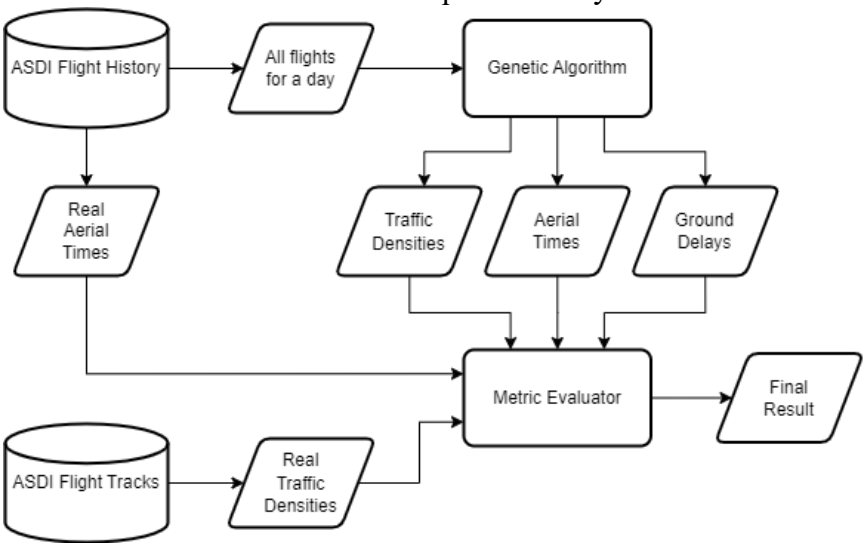


Figure 6.2 High Level Design for model evaluation

6.2 Modular Breakdown

The designs highlighted in the previous section demonstrate six modules for the product and an additional module for the evaluation of the model, and we detail all seven modules below.

6.2.1 Clustering Module

In this module, we utilize Spectral Clustering, an algorithm that uses graph theory and linear algebraic techniques to cluster a set of points. We obtain all the individual waypoints/fixes from the ASDI Waypoints data and form an input data table consisting of these waypoints' coordinates. We then input this table to the clustering module, which assigns each row of this table, or a waypoint, to a specific cluster. Each cluster is represented by a number ranging from 0 to N, N being the number of clusters, a parameter to the algorithm. We output a data table that maps each waypoint to a cluster.

6.2.2 Convex Hull Module

A convex hull is a closed geometric figure that bounds a set of points, such that the area of the Figure is minimum. Numerous convex hull algorithms are available, such as Graham Scan, Jarvis, and Chan. We choose Chan's algorithm due to better time and space complexity. The data table output by the sectorizing module is taken as an input, and the algorithm formulates a convex hull out of each cluster. A list of two-dimensional points represents the hull, so one obtains the geometric Figure by connecting these points. Therefore, the algorithm outputs N lists of 2D points, each corresponding to a specific cluster's convex hull.

6.2.3 Graph Generation Module

With the set of convex hulls from the previous module as an input, we generate a graph, with each hull being a vertex and adjacent hulls being connected by an edge. This graph is the framework upon which we run the genetic algorithm since any path generated must essentially pass through a sequence of vertices in the graph. This graph is called a traffic network since each vertex has information about its traffic.

6.2.4 Genetic Algorithm Module

Using the graph generated in the previous module, we develop a parallel genetic in CUDA. We define a fitness function that minimizes the path length and uses a mathematical technique to balance flight delays and traffic congestion. We develop a unique parallel crossover operator with time-varying chromosomes and an efficient repair function. We illustrate the method used to define a flight path that considers time and speed. This module also takes as input a set of input flights from the website and outputs the flight plan and delay. The output is given to the simulator and the website.

6.2.5 Simulator Module

We develop a simulator that shows all the flights progressing on the map as time passes. The user can interact with the simulator, pausing and restarting if required. The simulator plots the flight paths output by the genetic algorithm, considering when and where a flight will be on its path using its cruise speed.

6.2.6 Website Module

We develop a website using a client-server architecture that allows the end-user to input a set of flights using an ergonomic user interface and allows them to execute the CUDA genetic algorithm at the click of a button. The backend converts the input from the front end into a format that the algorithm can understand and executes the application. The backend also handles the conversion of the algorithm's output into a form that can be displayed on the front end. The website also provides an option to execute the simulator at the hit of a button.

6.2.7 Metric Evaluation Module

To evaluate our model's solution, we input all the US domestic flights that took place in a day to the website using the flight history table. We calculate the traffic densities for all sectors, flight times, and airport traffic from our result and compare it with their counterparts obtained using the actual tracks taken.

Chapter 7

IMPLEMENTATION

7.1 Implementation Timeline

Implementation of the project started in June 2022, wherein we followed a specific timeline of operations as detailed below:

Month	Year	Event
June	2022	Dataset Cleaning, Pre-processing
July	2022	Prototype of a GA in Python for Proof-Of-Concept
August	2022	Development of the GA in CUDA – Version 1.0, Website
September	2022	Development of the GA in CUDA – Version 2.0
October	2022	Development of the GA in CUDA – Version 3.0, Testing and Metric Analysis
November	2022	Documentation, Report Making, Presentation, and Thesis Submission
December	2022	Publication

7.2 Implementation Methodology

We first clean the dataset by dropping duplicates, converting all the spherical coordinates to corresponding cartesian counterparts, correcting the airport database to account for every airport in the USA, and getting the data ready for the sectorizing and graph generation modules. We then proceed to the Pre-Processing stage, wherein we cluster the cleaned waypoint list using a clustering technique. Here we introduced the concept of zones, centres, and sectors, where we modelled each cluster as a sector using a convex hull algorithm. We then form a graph out of the sector list, which would serve as the foundation upon which our GA would run. We implement the GA in Python as a prototype. After observing favourable output metrics, we parallelize the algorithm and utilize the GPU to decrease the degree of granularity of the parallel program in favour of good performance. We have written the final code in CUDA and C.

7.2.1 Dataset Cleaning

Three main files constitute the dataset: the Flight Tracks, Waypoints, and Flight History data tables. Each of these files is provided by the Aircraft Situation Display to Industry (ASDI) service by the Federal Aviation Authority (FAA) for August and September 2013. The Flight History data table has information regarding every single flight that took place in the months specified above, with each row corresponding to a flight and each column providing the departure and landing airport for the flight, the estimated and actual departure and arrival times, and a unique ID that identifies every flight throughout each data table. This table had many rows with null values, especially in the departure and arrival times columns; hence, we just dropped these rows because data loss was insignificant. There was no way to obtain the departure and arrival times, which were null since we could not extrapolate these values from previous occurrences for the flight. We also converted all the timestamps provided in the departure and arrival times columns to UTC since the data recorded was at local time.

The Flight Tracks data table has each flight's coordinates and ground speed in the Flight History table, recorded at one-minute intervals from departure to arrival. Each row corresponds to a particular timestamp at which data for some flight was recorded, with the columns being the ground speed, latitude, and longitude of the flight recorded for that specific timestamp. This table did not require cleaning as the timestamps were already in UTC. We directly used the table after dropping some columns we deemed unnecessary for the project, such as 'callsign' and 'altitude.'

	latitude	longitude
0	39.866667	-75.233333
1	39.866667	-75.150000
2	39.783333	-75.116667
3	39.783333	-75.066667
4	39.783333	-75.050000
...	...	...
3817923	45.250000	93.700000
3817924	45.300000	93.800000
3817925	45.516667	94.233333
3817926	45.816667	94.866667
3817927	33.583333	80.866667

3817928 rows × 2 columns

Figure 7.1

Coordinates for all unique waypoints

Each flight plan filed to the FAA always consists of a sequence of airspace/fixes, also called waypoints which are navigational aids that allow a pilot to navigate the airspace between the departure and arrival airports accurately. Every flight in the Flight History table files for a set of four to six flight plans and follows through with one, and for each of the filed flight plans, for each flight, we have the sequence of waypoints to be followed in the Waypoints data table. Each row of this table corresponds to a waypoint traversed at some point in a flight plan. Hence, the column headers are the latitude and longitude coordinates of a waypoint, the sequence number, and the ID of the flight plan. We extract all the rows with unique values for the latitude and longitude coordinates fields (Figure 7.1), extracting every waypoint in the table. For the two tables mentioned above, we needed a method to project these spherical coordinates to a flat surface or an XY plane; hence, we used a Python library called Basemap, which provides such a

conversion factor. Using this factor, we can convert all spherical coordinates (latitude and longitude) to their cartesian counterparts (easting and northing) and replace them in the two data tables as the final part of dataset cleaning. As observed in Figure 7.2, the Waypoints are scattered all over the globe; hence, we filter out those waypoints that lie above the USA, resulting in Figure 7.3.

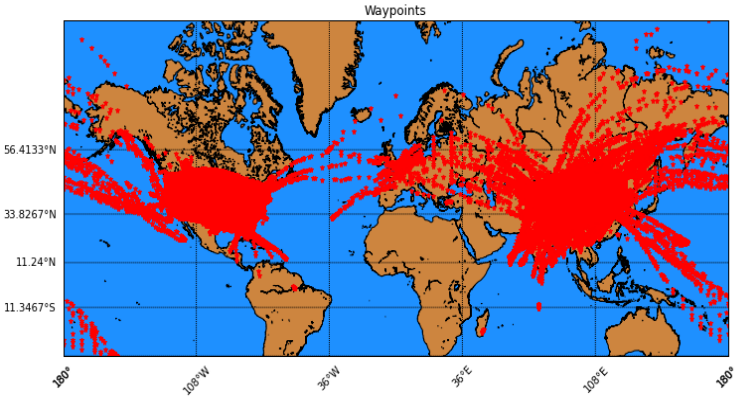


Figure 7.2 All Waypoints projected on a map

We now focus on getting the coordinates of all airports referenced in the Flight History data table. For this purpose, we use a JSON file obtained from an open-source GitHub repository, which maps every airport in the world (identified by its ICAO code) to its spherical coordinates, and we create a data table with every airport's ICAO code, name, latitude, and longitude. The table consisted of a total of 1103 airports in the USA. Figure 7.4 shows a plot representing all the airports and waypoints, proving that all airports lie within the boundary established by the waypoints.

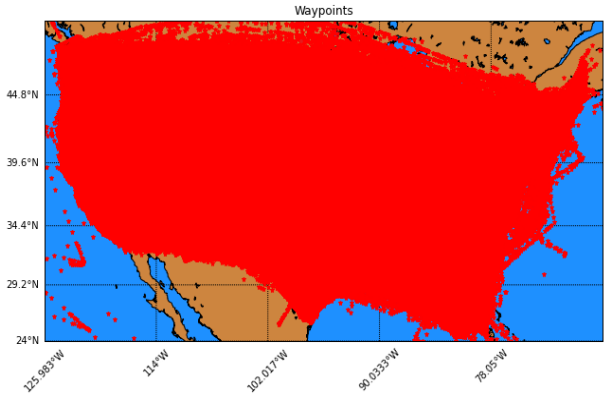


Figure 7.3 All Waypoints in the USA

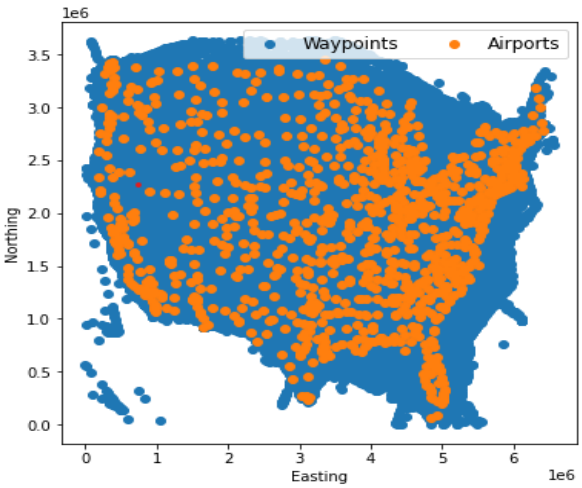


Figure 7.4 Airports and Waypoints

7.2.2 Pre-Processing

We first define the concept of zones, centers, and sectors for the pre-processing stage. By plotting the set of US waypoints obtained in the previous step on a cartesian plane (Figure 7.4), we can observe that the extent of the easting values ranges from zero to six million. Hence, we divide the waypoint data into six parts based on its easting coordinate, with each waypoint belonging to a particular 'zone.' A zone stretches across one million in easting values; hence, there are six such zones. We divide each 'zone' based on northing extent into four to six 'chunks' each, such that each 'chunk' has approximately a hundred thousand waypoints. The reason for such an empirical division

is to mitigate the time required to cluster the waypoints since most clustering algorithms are linearly dependent on the number of data points given as input.

We cluster each such chunk into five 'centres' each using Spectral Clustering, which is an inbuilt algorithm module in the sci-kit learn library in Python; the output is represented in Figure 7.5, which shows each of the twenty-five chunks being five clusters each, resulting in one hundred and twenty-five clusters. We further cluster each of these centres using Spectral Clustering into five clusters/'sectors' each, resulting in 1250 clusters or sectors. These sectors will serve as the zones of the zonal traffic network we will establish over the USA, upon which we run the algorithm. We represent the final output of the sectorization module in Figure 7.6, which shows 1250 sectors, each represented by a unique colour.

For the next step in our pre-processing, we implement Chan's convex hull algorithm in Python, which takes in as an input a set of points, and outputs those points corresponding to a convex hull's boundary built around it. We use this algorithm to obtain the convex hulls for every sector generated in the previous step, thus resulting in 1250 convex hulls. Figure 7.7 shows a fragment of the entire image where we model sectors as hulls and indicates the airports being part of specific hulls. The resulting airport-to-sector mapping is extracted and stored as it will be helpful for the genetic algorithm.

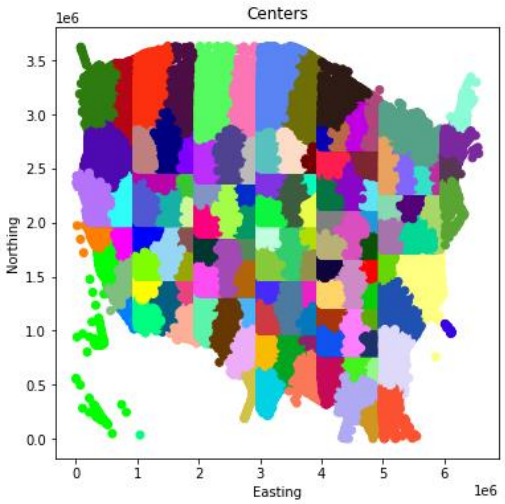


Figure 7.5 Chunks and Centres

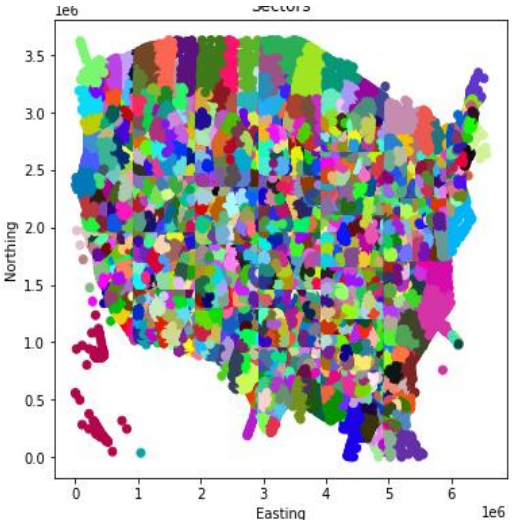


Figure 7.7 Convex Hulls

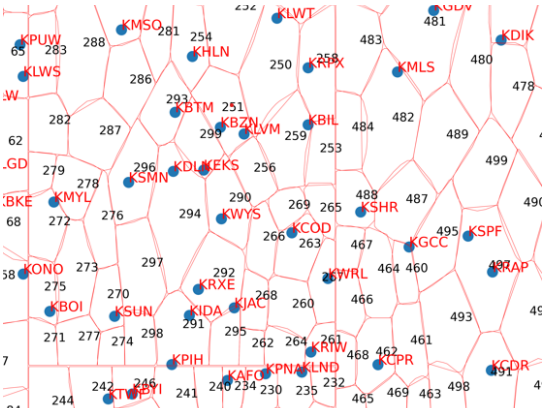


Figure 7.6 Sectors

The set of hulls obtained in the previous step are right next to each other, as seen in the adjoining Figure, and hence we can construct a graph from it by joining two neighbouring hulls with an edge. The graph thus models each hull as a vertex. We use the 'Shapely' library in Python to determine whether two hulls are adjacent to each other (and hence have an edge between the corresponding vertices in the graph). Shapely is a computational geometry library that allows us to

model each hull as a Polygon object and provides us with a method, `distance ()`, that finds the Euclidean distance between any two such Polygon objects. We use this method to get the distance between all possible pairs of hulls and empirically determine that two hulls are adjacent if the distance between them is less than or equal to ten thousand meters. The resulting graph had 1250 vertices and 3725 undirected edges. We store this graph as an igraph object using the 'igraph' graphing library for Python and R. We show the sectorized US map in Figure 7.8 where the states are shown in black and the sectors or hulls are shown in red.

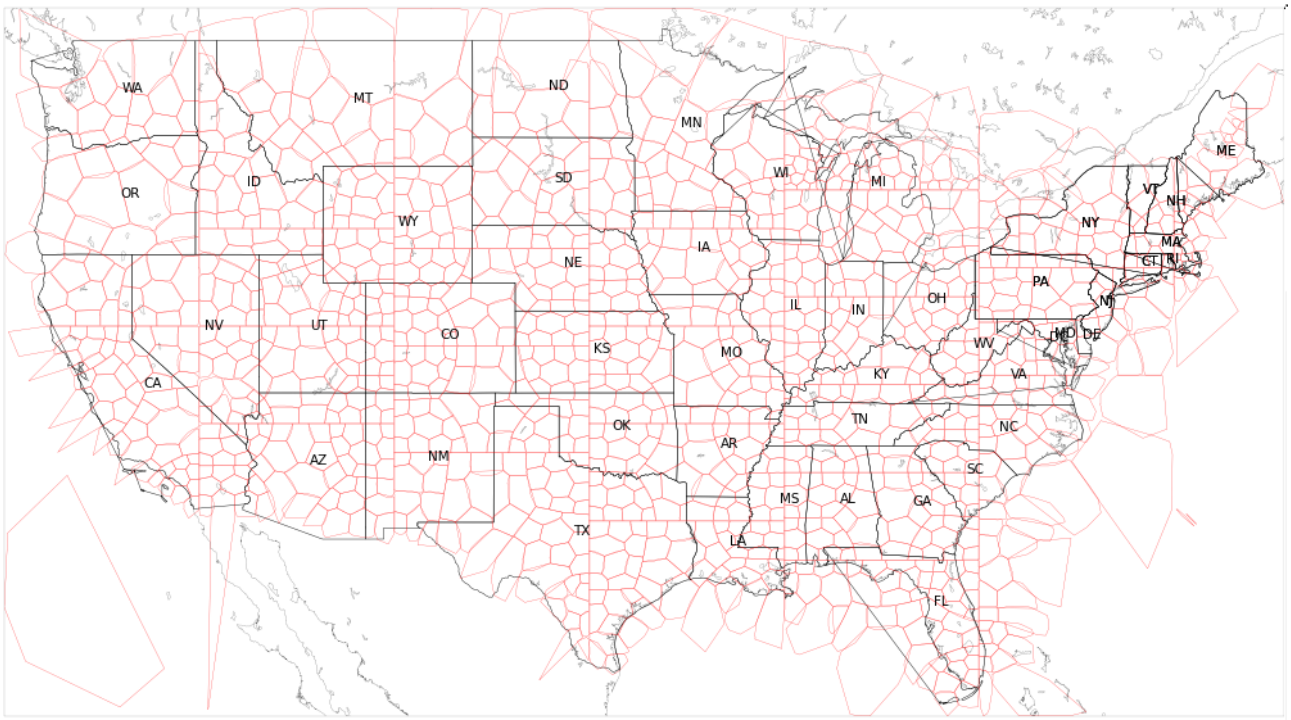


Figure 7.8 Complete USA map with the sectors

We now extract the 'connecting point' between every pair of adjacent vertices; the genetic algorithm will use these 'connecting points' for pathfinding, and we will present the exact pathfinding method in Section 7.2.4. We first need to find the centroids of every convex hull/sector. We do so by representing each hull as a Shapely Polygon and extracting its centroid using the built-in calculator that stores the polygon's centroid as a property of the object. As illustrated in Figure 7.9, we join the centroids of two adjacent sectors/hulls using a line and find its intersection point with the actual hull boundary.

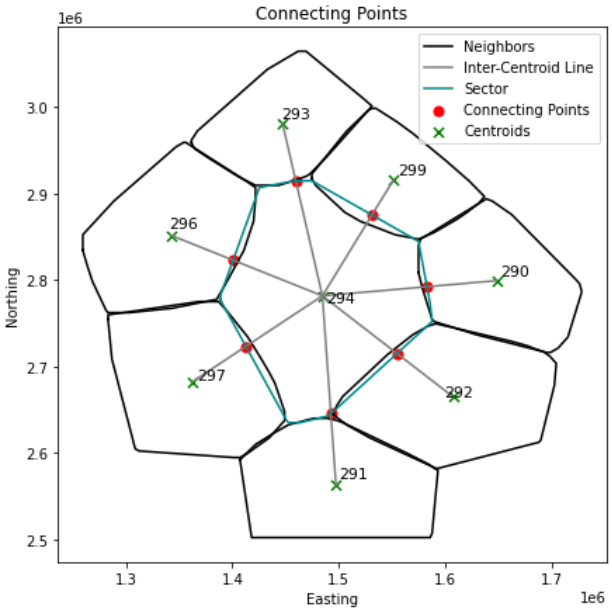


Figure 7.9 Connecting Points

We do this operation using the Shapely library by representing the inter-centroid line and the convex

hull as two LineString objects and then finding the intersecting point between them using the `intersection()` method. We carry out this operation on every pair of adjacent sectors and obtain a set of 7788 intersecting points, two points per edge, converting the graph into a directed graph. We store each intersecting point as an edge attribute in the igraph object and write the whole graph into a text file as an adjacency list which would serve as an input to the CUDA module.

7.2.3 GPU-Accelerated Genetic Algorithm

We now implement a parallel genetic algorithm in CUDA and present the various kernels involved and the parallelization strategy followed. We begin by detailing how a basic genetic algorithm works, then provide a high-level overview of CUDA and the motivation for using a GPU-Accelerated algorithm over a serial program. We then discuss the pathfinding method and the method used to ensure the routes generated to find a balance between path cost and path traffic. We then detail how the algorithm can optimally schedule flights to ensure we obtain a complete solution to the Air Traffic Management Problem.

7.2.3.1 Introduction to Genetic Algorithm

A genetic algorithm is an optimization technique that belongs to the metaheuristic algorithms category, explored in recent literature. Other algorithms in this category include particle swarm optimization, ant colony optimization, tabu search, and many other techniques that mimic biological processes. These algorithms aim to optimize a particular metric or, mathematically speaking, find the minima or maxima of a multivariate function. These functions usually have more than three unknowns. Using classical calculus to find the exact combination of values for all the variables that lead the function to the global minima or maxima may be time-consuming and indeterministic.

All metaheuristic algorithms, essentially, find these optima by experimenting with various values for the variables and propagating the solutions that drive the function to the desired optima. It is like gradient descent followed by artificial neural networks, which utilize a more mathematically rigid technique by plotting a locus of a point along the function plane and using partial derivatives to find out the direction which leads to the optimal value, aided by a learning rate parameter that determines the distance to travel in that direction (Figure 7.10). On the other hand, Metaheuristic algorithms plot loci of multiple points along the function plane and, using some form of a coordination mechanism between the points, allow them to collect at one point on the function plane; this is the convergence

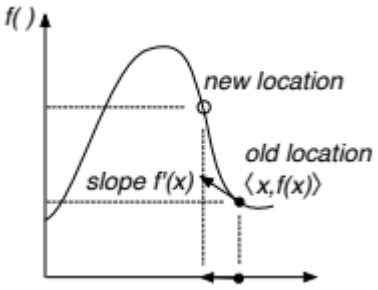


Figure 7.10 Gradient Ascent

point where the algorithm has concluded that the function has reached an optimum; whether this is the global optima or local depends on the input parameters.

A genetic algorithm mimics evolutionary mechanisms in biological systems, wherein a chromosome represents an organism, a candidate solution to the target function. A population is a collection of chromosomes or a set of candidate solutions to the target function, with the population size being one of the input parameters.

The algorithm first generates an 'initial population' by randomly generating chromosomes and then performs four operations: Selection, Crossover, Mutation, and Elimination, on the population in a loop, and each loop iteration is called a generation. Each chromosome has a specific 'fitness' value which is the evaluated value of the target function. The chromosome structure must, therefore, allow it to carry the values for the variables that make up the multivariate target function or at least enable the calculation of these values on the fly. The algorithm reaches the convergence point when all chromosomes become similar and give the same set of values for the variables of the target function. At this point, the loop terminates, and the algorithm outputs the values for the variables that optimize the target function. We depict the entire process in a flowchart in Figure 7.11. We present the detailed algorithm in section 7.2.3.2.

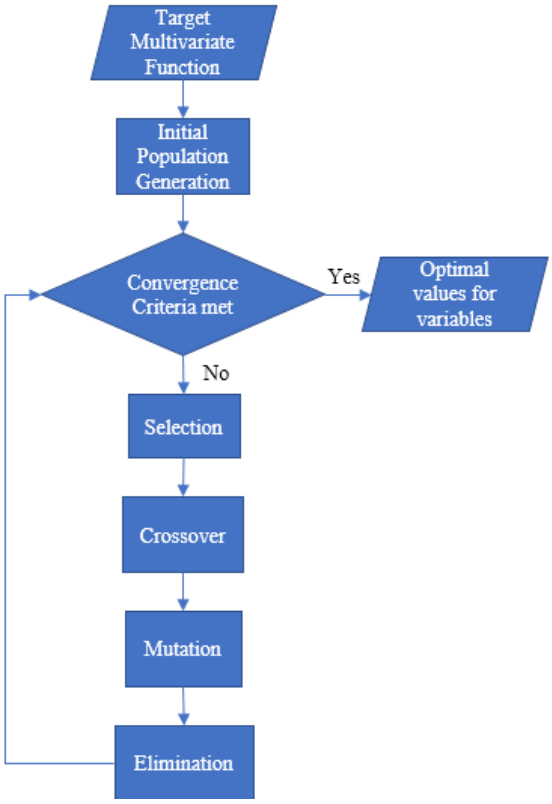


Figure 7.11 Genetic Algorithm Process

7.2.3.2 Overview of CUDA

The language of choice for implementing the algorithm is C due to its known performance advantages over other languages. We had to focus primarily on performance because we had to run the program for all the flights in an entire day to gain an accurate metric for the result. Due to the GA being an inherently slow process, as we witnessed when we developed the prototype in Python, we decided to parallelize the algorithm and utilize the GPU to accelerate it further. The final algorithm

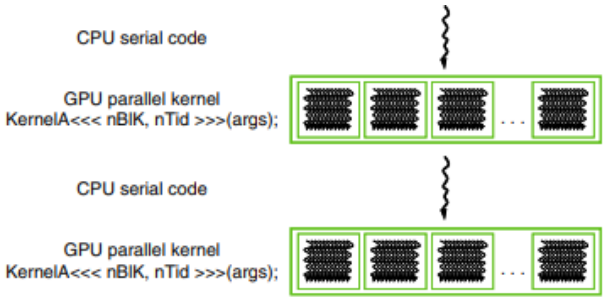


Figure 7.12 CUDA Execution model

was high-performance and could converge within half a second when tested on Nvidia GeForce RTX 3050 GPU.

Compute Unified Device Architecture (CUDA) is an API provided by Nvidia Corp. that seamlessly integrates with many programming languages like C and Java. It allows one to split their codebase into 'host' and 'device' codes, with the CPU executing the host code and the GPU running the device code, also called 'kernels.' The code file must have a .cu extension, and we use the Nvidia CUDA Compiler (NVCC), a proprietary compiler by Nvidia intended for use with CUDA to compile it. The compiler is available as part of the Nvidia CUDA Toolkit. Following the classical procedural programming paradigm, the device and host codes are separate functions, with the host functions calling the device function via a mechanism called a 'kernel launch.' (Figure 7.12). The kernel calls are always asynchronous; the CPU does not wait for the GPU to finish its task and moves on to execute further statements. The GPU runs the kernels in the order in which they are called, as they are placed in the same default stream.

A GPU consists of multiple streaming multiprocessors (SMs), each composed of numerous streaming processors (SPs), also called CUDA Cores. The GPU we use is the Nvidia GeForce RTX 3050, which has 20SMs, each with 120 SPs, totalling 2560 SPs or 2560 CUDA Cores. These processors are different from traditional CPUs because they do not have a large cache/SRAM, allowing many processors to fit on the dye and allowing the GPU to spawn many threads. The threads follow a hierarchy, with a bunch of threads being part of a block and a bunch of blocks being part of a grid, and follow a memory hierarchy, as shown in the adjoining image (Figure 7.13), with global memory, block memory, and thread-local memory. The GPU is suited mainly for SIMD programming wherein each thread performs the same operation but on different inputs, which applies to our use case due to the repetitive nature of the procedures involved.

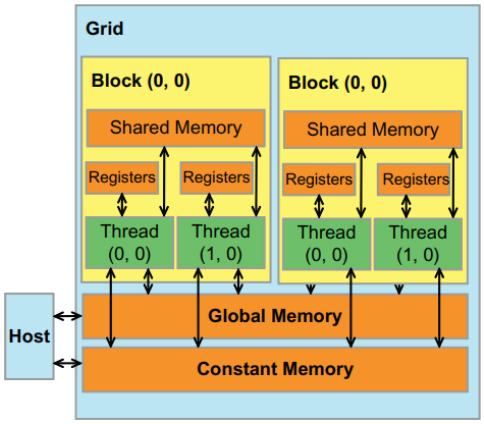


Figure 7.13 GPU Memory Hierarchy

The kernels are of two types, global kernels, and device kernels. The GPU calls the device kernels and is like a function call but executed on the GPU. The CPU calls the global kernels by specifying launch parameters within a triple angular bracket pair which signifies the number of blocks to spawn (`gridDim`) and the number of threads within each block (`blockDim`). Each block has a unique block ID (`blockIdx`), and each thread within it has a unique thread ID (`threadIdx`). A combination of both these IDs uniquely identifies any thread across all the blocks.

A general method followed for the launch parameters is `<<<(NumThreads/32) + 1,32>>>`where NumThreads is the number of threads spawned in the GPU to execute the kernel. Within the kernel, we use `threadIdx.x + (blockIdx.x * blockDim.x)` to identify the executing thread and hence identify the pieces of data in the global memory it is allowed to access. The CUDA model thus implements data parallelism by having the same kernel code executed simultaneously by numerous threads but on different data, as referenced by the thread identity.

7.2.3.3 Path Fitness Evaluation Strategy

We have implemented the genetic algorithm for pathfinding in this work. In this section, we detail the customizations to a classical GA to suit the purpose of pathfinding, the target function, and the various operations involved.

A path across the graph generated in the pre-processing stage stretches from the sector containing the source airport, ends at the sector containing the destination airport, and follows a series of sectors or vertices in the underlying graph (`SectorPath`), as shown in Figure 7.14a, b. The first variable to minimize would be the path length (`L`) itself because a longer path would result in more time spent in the air (aerial time) by the aircraft, consequently driving up the amount of fuel consumed, resulting in a higher economic cost. Since the path is a sequence of sectors, we join all the corresponding connecting points with straight lines and aggregate their lengths (evaluated using the Euclidean distance) to calculate the path length as depicted in Figure 7.14c. We use the cartesian coordinates of the source and destination airports as the first and final connecting points, respectively.

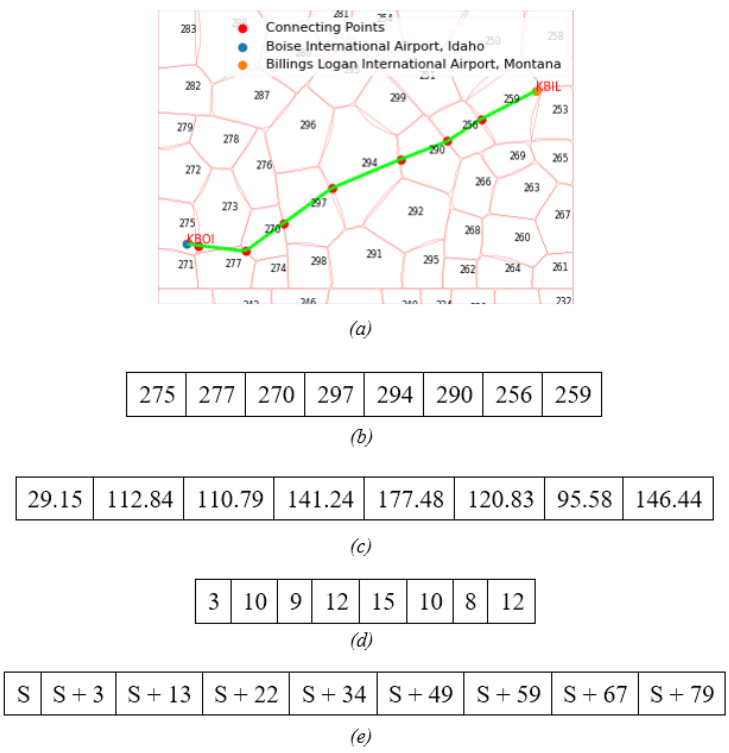


Figure 7.14 Path Representation Example

For example, a flight from Boise International Airport, Idaho (KBOI), to Billings Logan International Airport, Montana (KBIL), is depicted in Figure 7.14(a). The flight follows a sequence of sectors, which we can observe in its flight path and represent as an array (b). We also depict the sequence of connecting points joined by straight lines in the Figure, with the two airports being the starting and final connecting points. The number of lines drawn to join the points will hence be equal

to the number of sectors, and we depict this in the Figure where we draw eight lines and note down their lengths in (c).

These lengths are in kilometres, and we calculate them using the Euclidean distance formula. We take as an input the cruise speed of the aircraft, which we assume to be constant throughout the flight, and use the classic distance formula $time = \frac{distance}{speed}$, to calculate the amount of time spent in each sector. We fix the unit of time as minutes and the unit of distance as meters, making the unit for speed as meters per minute. Since cruise speed is usually measured as ground speed and given in knots, we use a conversion factor to convert the cruise speed to meters per minute (1 Knot = 30.8667 Meters per minute). Therefore, we get a `TimingArray`, depicted in (d), where each value corresponds to the rounded value of the respective distance divided by the speed. Hence, we measure how many minutes an aircraft spends in each sector it passes through in its path, aiding us in traffic management.

In the Figure, we take the speed (ground speed) to be 400 knots, which is 12346.7 meters per minute, and the distance was measured in meters, therefore giving the time array in minutes. We take the rounded value obtained from the division for simplicity and assume the cruise speed to be constant throughout the flight. Finally, we make a prefix sum array of the `TimingArray` in (e), called (`CumulativeTimingArray`) where S represents the actual runway departure of the aircraft. Hence the plane traverses the `i`th sector in its path from the `i`th minute to the `(i + 1)` th `minute` in the prefix sum array. For example, if we take S to be 125, which means the aircraft's runway departure time is 2:05 A.M., then it would traverse the sector 270 `(i = 2)`, from minute 138 to minute 147, or from 2:18 A.M. to 2:27 A.M., which is a duration 9 minutes, present at the `(i = 2)` th index of the `TimingArray` (c).

We maintain a matrix called `TrafficMatrix` with 1250 rows, where 1250 is the total number of sectors, and 2880 columns, where 2880 is the number of minutes in two days. This matrix captures the number of aircraft present in any sector at any minute of the day. Hence `TrafficMatrix[i][j]` will be the number of aircraft present in the `i`th sector in the `j`th minute, and we also maintain a sum of all the values present in this matrix called '`TrafficSum`.' For every flight we provide a path for in the GA, we update the matrix as well; for example, for the path in the Figure, we would add one to columns 138 to 146 inclusive for the 270th row of the matrix. This is done for all the sectors in the `SectorPath` using the `TimingArray` when the path is output as a solution for a flight to follow.

While generating a path for a flight, to evaluate how much traffic it encounters along its path, we take the sum of all the corresponding `TrafficMatrix` values and call the result as `TrafficFactor`, for example, while considering the path in the Figure 7.14b, we take S as 125 and hence evaluate `TrafficFactor` for that path as so.

`TrafficFactor = sum(TrafficMatrix[275][125…127]) + sum(TrafficMatrix[277][128…137]) + sum(TrafficMatrix[270][138…147]) + … + sum(TrafficMatrix[259][192…204].`

Hence, `TrafficSum-TrafficFactor` would represent the amount of traffic avoided in the flight path and will be the second parameter (to maximize) in the target function.

Every flight has a scheduled runway departure time fixed by the carrier and is the tentative time at which the flight should depart. If the amount of traffic faced, `TrafficFactor`, is very high for the path with the shortest length, the airplane will prefer a longer route with lesser `TrafficFactor`. On the other hand, as a longer path would mean higher economic costs, one might consider delaying the flight at the departure airport or imparting a 'ground delay' to it. This delay would allow the plane to take a shorter path with lesser `TrafficFactor` by departing when the airspace has relatively freed up while compromising passenger satisfaction. Hence the air traffic management system should compare the two scenarios, one in which the flight departs at the scheduled time or with some delay, and pick the best alternative.

An optimal trade-off would be a significant decrease in the traffic encountered in the path (`TrafficFactor`), with the shortest possible delay added to the flight. To evaluate this mathematically, we assess the `TrafficFactor` for every minute of delay added (with the maximum possible flight delay being 60), resulting in a graph as shown in Figure 7.15. We represent this graph in the code by an array of size 60, `TrafficArray`, where `TrafficArray[i]` is the `TrafficFactor` calculated for the `ith` minute of flight delay. We now proceed to find the set of minima in this graph which we illustrate using blue dots in the charts in Figure XX and store in a set called `MinimaSet`. We then draw lines from `TrafficArray[0]` to all the minima points in `MinimaSet` and calculate each line's slope, and we represent these as dashed blue lines in the Figure. Finally, we

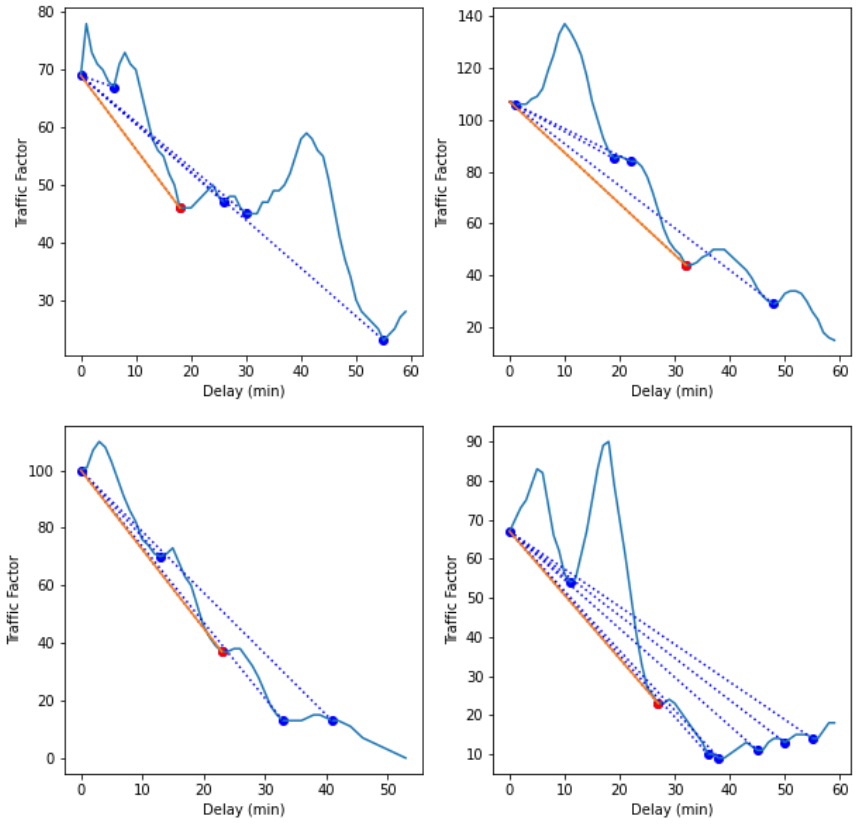


Figure 7.15 Traffic Factor and Delay

choose the line with the least slope (which we show as a solid red line in the Figure), hence the idea being to select the minima that would provide the most significant reduction in the y value from `TrafficArray[0]` for a minor increase in the x value from 0.

The optimal delay is the third parameter we would evaluate while calculating the fitness of a solution in the GA, where we output the optimal delay for a flight path, which corresponds to the minima that offers the least slope, as demonstrated. In the Figure, the delays at which the red dots occur are those minima that offer the least slope and hence are the `OptimalDelay` returned for each.

For a flight path to be considered good, the angular deviation experienced by it must be less, as too many sharp turns would be detrimental to the aircraft's passengers' satisfaction. To evaluate the angular variation in the path, we join the series of Connecting Points that it follows. We then find the exterior angle between every pair of adjacent lines and output their average as the `AngularCost` of the path. We represent this diagrammatically in Figure 7.16 where we show the Connecting Points as red dots and the path in blue, and the angular deviations as θ. The final `AngularCost` is the average of all such angular deviations and would serve as the fourth parameter to be evaluated in the fitness function.

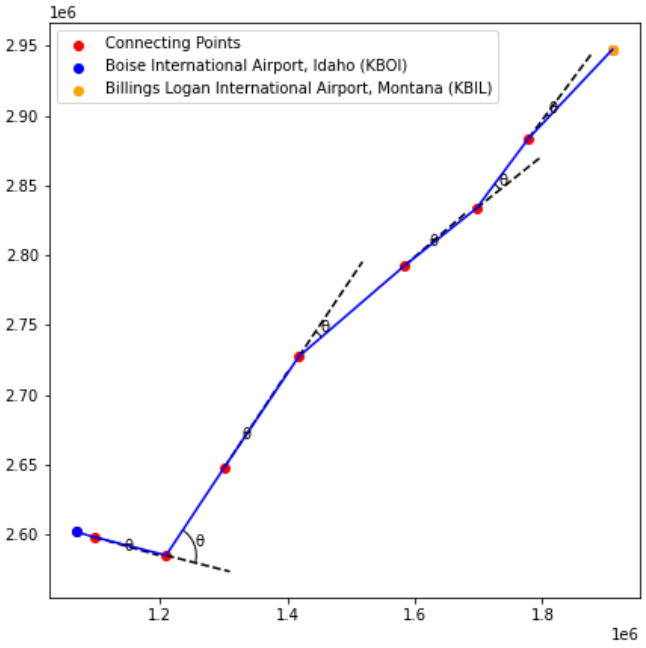


Figure 7.16 Path Angular Deviations

Every airport has, on average, two runways that can be used for take-off and landing by airplanes. The departing or arriving aircraft blocks the runway for a fixed time, during which no other aircraft can use the same runway. Requests by other aircraft to use a blocked runway are handled by either allocating it to another runway or delaying its departure or arrival until the blocked runway is freed. Our model assumes that every flight occupies or blocks a runway for one minute. Since the generated path allows us to know the precise minute of departure and arrival, we should ensure that the number of aircraft departing or arriving at any specific minute of the day at any airport is always lesser than or equal to the number of runways they have. For example, KBOI has two runways, and KBIL has three runways; hence, at the max, KBOI can handle two departures or arrivals in the same minute, and KBIL can handle three departures or arrivals in the same minute.

The above constraint is effectively handled by using an `AirportMatrix` with N rows and 2880 columns, where N is the number of airports in the USA (approx. 1103), and 2880 is the number of

minutes in two days in a similar fashion as the `TrafficMatrix`. Therefore, `AirportMatrix[i][j]` is the number of runways used concurrently or the number of aircraft serviced in the i^{th} airport in the j^{th} minute.

We obtain the number of runways available at each of the N airports and ensure that the constraint `AirportMatrix[i][j] <=` R_i is satisfied while generating a solution. Here R_i is the number of runways available in the i^{th} airport.

With this, we can formally define the fitness function and the constraints involved, which would be used to evaluate the quality of a solution in the genetic algorithm. Here we input a flight path as a sequence of sectors (`SectorPath`) from the departure to the arrival airport, the aircraft's cruise speed, the `TrafficMatric` and `AirportMatrix`, and the underlying `SectorGraph` with the Connecting Points. The output is the Fitness of the path as a floating-point number, an integer representing the optimal amount of delay to be added to its `ScheduledDepartureTime` (which we denote as the N^{th} minute from 0:00), and the `TimingArray` as well, which represents the time spent (in minutes) spent in each sector of `SectorPath`.

```
Algorithm 1
        Path Fitness Evaluator
Input
        SectorPath, Graph, TrafficMatrix, AirportMatrix, ScheduledDepartureTime (T),
        CruiseSpeed, DepartureAirport (D), ArrivalAirport (A).
Output
        Fitness, OptimalDelay, TimingArray
Code
        TimingArray=[]
        DCoords=D.getCartesianCoordinates()
        PrevPoint=DCoords
        ConPointArray=[DCoords]          // Used for AngularCost
        L=0
        for i ∈ {0 … SectorPath.length()-2}
                CurSec = SectorPath[i]
                NextSec = SectorPath[i+1]
                NextPoint = Graph.getConnectingPoint(CurSec,NextSec)
                ConPointArray.append(NextPoint)
                Distance = getEuclideanDistance(PrevPoint,NextPoint)
                TimingArray.append(Distance/CruiseSpeed)
                L += Distance
                PrevPoint = NextPoint
        NextPoint = A.getCartesianCoordinates()
        Distance = getEuclideanDistance(PrevPoint,NextPoint)
```

```
TimingArray.append(Distance/CruiseSpeed)
L+=Distance
ConPointArray.append(NextPoint)
ArrivalTime=sum(TimingArray)+ScheduledDepartureTime
AngularDeviation=0
for i ∈ {0 … ConPointArray.length()-3}
      AngularDeviation+=getAngle(ConPointArray[i],ConPointArray[i+1],ConPointArray[i+2])
AngularCost= AngularDeviation/ConPointArray.length() //Average Angular Deviation
StaticCost=(180-AngularCost)/L
TrafficArray=[]
for delay ∈ {0 … 60}
      D_T=AirportMatrix[D][T+delay]
      A_T=AirportMatrix[A][ArrivalTime+delay]
      R_D=D.NumberOfRunways
      R_A=A.NumberOfRunways
      TrafficFactor=0
      if(D_T >= R_D or A_T >= R_A)
            TrafficArray.append(sum(TrafficMatrix))
      Else
            CurrentSector=SectorPath[0]
            for time ∈ {T+delay … TimingArray[0]}
                  TrafficFactor+=TrafficMatrix[CurrentSector][time]
            PrevTime=TimingArray[0]
            for SectorNumber ∈ {1 … SectorPath.length()}
                  for time ∈ {PrevTime … TimingArray[SectorNumber]}
                        TrafficFactor+=TrafficMatrix[SectorNumber][time]
                  PrevTime=TimingArray[SectorNumber]
            TrafficArray.append(TrafficFactor)
OptimalDelay=getOptimalDelay(TrafficArray)
Fitness=StaticCost x (sum(TrafficMatrix) – TrafficArray[OptimalDelay])
return {Fitness, OptimalDelay, TimingArray}
```

The `getOptimalDelay(TrafficArray)` is the function that returns the index in `TrafficArray` where the optimal delay occurs as per the minima method discussed previously. The other methods used above are self-explanatory.

7.2.3.4 Parallel Genetic Algorithm

The Parallel Genetic Algorithm used to solve the Air Traffic Management Problem consists of five parallel algorithms executed one after the other until convergence. The parallel GA generates

one path it considers the fittest for a flight; hence, it is executed N times for N input flights. We detail the procedure involved in the GA in this section.

7.2.3.4.1 Parallel Initial Population Algorithm

The first step in the algorithm would be to generate an initial population, a random set of solutions to the problem, which, in our case, is a set of random paths from departure to arrival airport. The paths are represented as discussed in the previous section, each path being a sequence of sectors. For example, Figure 7.17 shows four random paths from KBOI to KBIL with four colors. This set of random paths generated forms the initial population. The size of the initial population (`P, Population Size`) is a parameter of the GA. Each path in the population is referred to as a 'chromosome.' Next, we run the path fitness evaluation procedure detailed in the previous section and obtain, for each path or chromosome, its `Fitness`, `OptimalDelay`, and `TimingArray`. The procedure to generate this initial population is crucial to the system's performance as we would require a large Population Size to reach a high-quality solution. We detail the algorithm followed to generate the initial population below.

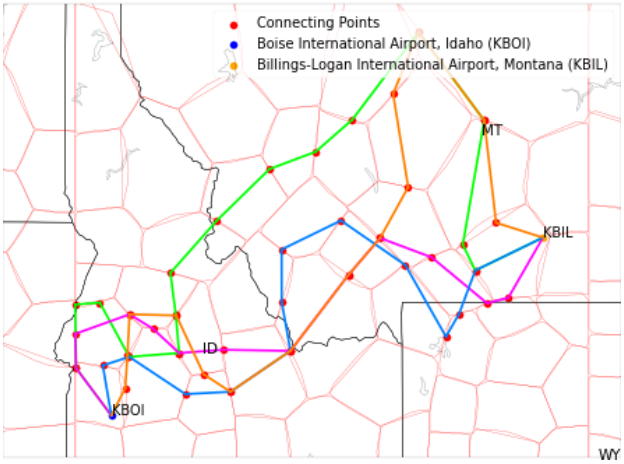


Figure 7.17 Random Paths generated for Initial Population

We obtain the sectors where the departure and arrival airports are present (`DepSec` and `ArrSec`, respectively) and run a randomized Depth-first search (DFS) on the `Graph` from source to sink. A randomized DFS continuously picks adjacent vertices randomly until the sink is reached. If a dead-end is encountered where no unvisited adjacent vertex exists, the whole DFS is restarted. The parallelization strategy for this algorithm is relatively easy, wherein one needs to spawn `P` threads on the GPU, where each thread performs its own randomized DFS and generates one chromosome. Hence a population of size `P` is achieved by generating one chromosome each by `P` threads. We chose a randomized DFS instead of a plain DFS because we observed a drastic improvement in algorithm performance since the graph resembles a grid.

The algorithm takes as input the `DepSec` and `ArrSec`, an empty array `FitnessArray`, to store the `Fitnesses` of `3P/2` chromosomes, an empty array `OptimalDelayArray` to store the `OptimalDelay` for `3P/2` chromosomes, an empty matrix (2D array) `TimingArrayMatrix` to store the `TimingArray` for the `3P/2` chromosomes and the `Graph`. The 2D array `Population` is also input with `3P/2` rows, and the paths or chromosomes are stored here. Although we generate only `P` chromosomes or paths in

this function, we necessarily keep `P/2` extra blank entries in all the above data structures since we will utilize them later.

```
Algorithm 2
	Parallel Initial Population Generator
Input
	DepSec, ArrSec, Graph, FitnessArray, OptimalTimeArray, TimingArrayMatrix,
	Population
Output
	-
Code
	parallel foreach ThreadID ∈ {0 … P-1}
		visited=set()
		gotPath=False
		while(not gotPath):
			visited.clear()
			visited.add(DepSec)
			CurrentSector=DepSec
			Chromosome=[DepSec]
			while(not gotPath):
				ValidNeighbors=Graph.neighbors(CurrentSector)) not in visited
				if(ValidNeighbors.size()==0):
					break
				NextSector=RandomChoice(ValidNeighbors)
				Chromosome.append(NextSector)
				visited.add(NextSector)
				if(NextSector==ArrSec):
					gotPath=True
					break
				CurrentSector=NextSector
		Population[ThreadID]=Chromosome
		Fitness, OptimalDelay, TimingArray=PathFitnessEvaluator(Chromosome)
		FitnessArray[ThreadID]=Fitness
		OptimalDelayArray[ThreadID]=OptimalDelay
		TimingArrayMatrix[ThreadID]=TimingArray
```

7.2.3.4.2 Parallel Selection Algorithm

The selection procedure selects good or "fit" chromosomes to be crossed over in the Crossover procedure. We utilize a random compare, and select algorithm that randomly chooses two chromosomes from the population, compares their `Fitnesses` and selects the chromosome with the higher `Fitness` value. This method is highly parallelizable and reduces selection pressure compared

to other selection algorithms like the roulette wheel. We set the `Selection Size(S)` as `P/2`, which is the number of chromosomes to be selected from the `Population`. The parallelization strategy followed is to spawn `S` threads, where each thread compares and selects between two chromosomes. We use an array of size `P` containing numbers in the range `[0 … P-1]`, called `SelectionPool`. This array will serve as the indexing mechanism for the threads involved. We demonstrate its usage in Figure 7.18, where two chromosomes are to be selected from a population of size four by two threads, T1 and T2. The `SelectionPool` array is accessed by the two threads, with T1 accessing the index `[0,1]` and T2 accessing the index `[2,3]` of the `SelectionPool` array, which indicates to the threads which chromosomes they must select from. As indicated in the Figure, T1 must select from C1 and C4 and T2 must select from C2 and C3. We must ensure to shuffle this `SelectionPool` randomly in every iteration of the GA.

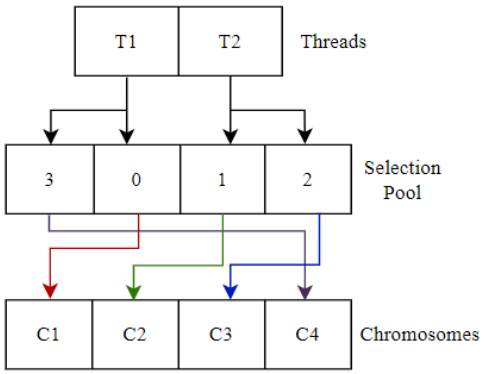


Figure 7.18 Usage of SelectionPool

The input is the `Population` array, the `FitnessArray`, an empty array of size `S`, `SelectedArray`, and `SelectionPool`. The `SelectedArray` holds the indices of the chromosomes selected.

```
Algorithm 3
	Parallel Selection Algorithm
Input
	Population, FitnessArray, SelectionPool, SelectedArray
Output
	-
Code
	parallel foreach ThreadID ∈ {0 … S-1}
		ChromosomeOneIndex=SelectionPool[2*ThreadID]
		ChromosomeTwoIndex=SelectionPool[2*ThreadID+1]
		if(FitnessArray[ChromosomeOneIndex]> FitnessArray[ChromosomeOneIndex])
			SelectedArray[ThreadID]=[ChromosomeOneIndex]
		else
			SelectedArray[ThreadID]=[ChromosomeTwoIndex]
```

7.2.3.4.3 Parallel Crossover Algorithm

The crossover procedure involves combining two “fit” chromosomes to try and achieve offspring chromosomes that combine the “good” parts of their parents and hence would be “fitter” than them. This procedure allows the members of the population to coordinate with each other to reach the

optimal value of the fitness function. The crossover strategy is as follows; we use a modified version of a single-point crossover, where the two parents interchange a part of their path, as shown in Figure 7.19. The problem is to figure out the `CrossoverPoint` at which the interchange occurs. In the Figure, the `CrossoverPoint` is the element D which occurs in both the parent chromosomes C1 and C2. The Offsprings are C3 and C4. We mark a sector in SectorPath as a CrossoverPoint if the aircraft will be in that sector is the same time period, if it takes either of the two parent paths. For example, in the Figure, if the plane takes the path C1 and is present in the sector D in the time frame [150, 160], and if the aircraft takes the path C2 and is present in sector D in the time frame [155, 175], we can conclude that in either case, the aircraft is in the sector D at the same time (for the time frame [155,160]) making D a `CrossoverPoint`.

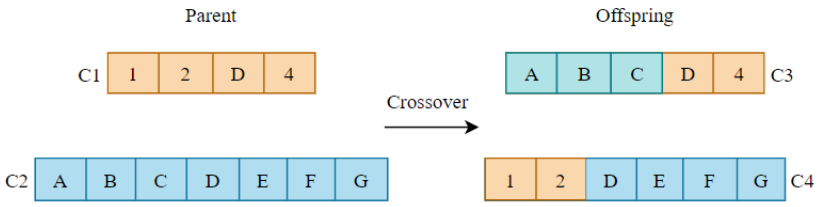


Figure 7.19 Timed Single Point Crossover

We make logical sense from this argument by stating that a crossover would thus be a diversion that the aircraft could carry out mid-air. The crossover procedure would hence, also consider time flow and flight delays. The offspring chromosome does not replace the parents in the population, and we add them to the `Population`, increasing the population size from `P` to `3P/2`. The chromosomes selected from the previous step, `SelectedArray`, are the parent chromosomes. We use a `ReplacementPool` here, an array of size `P/2`, initially containing numbers in the range `[P, (3P/2)-1]` and will be updated in further iterations of the GA. This array marks the indices of the chromosomes in the `Population` that the offspring chromosomes C3 and C4 must replace.

The parallelization strategy is similar to the Selection Algorithm, where we keep a `CrossoverPool` of size `S/2` and spawn `S/2` threads, where each thread would crossover two chromosomes indexed by the `CrossoverPool` in the `SelectedArray`. In addition, we utilize an efficient algorithm, `FindCrossoverPoints`, that takes the prefix sums of the two parents' `TimingArrays`, finds those common sectors between the two parents where overlapping time intervals exist and marks these sectors as `CrossoverPoints`. We also input `ReplacementPool` for storing the offspring.

```
Algorithm 4

        Parallel Crossover Algorithm
Input

        Population, FitnessArray, CrossoverPool, SelectedArray, TimingArrayMatrix,
        ReplacementPool, OptimalDelayArray
Output
```

```
	-
Code
	parallel foreach ThreadID ∈ {0 … (S/2)-1}
		C1Index=SelectedArray[CrossoverPool[2*ThreadID]]
		C2Index=SelectedArray[CrossoverPool[2*ThreadID+1]]
		C3Index=ReplacementPool[2*ThreadID]
		C4Index=ReplacementPool[2*ThreadID+1]
		C1= Population[C1Index]
		C2= Population[C2Index]
		C1TimeArray=PrefixSum(TimingArrayMatrix[C1Index],OptimalDelay[C1Index])
		C2TimeArray=PrefixSum(TimingArrayMatrix[C2Index],OptimalDelay[C2Index])
		CrossoverPoints=FindCrossoverPoints(C1, C2, C1TimeArray, C2TimeArray)
		ChosenPoint=random.choose(CrossoverPoints)
		C3,C4=PathPartInterchange(C1,C2,ChosenPoint)
		Population[C3Index]=C3
		Population[C4Index]=C4
		for CIndex in [C3Index,C4Index]:
			Child= Population[CIndex]
			Fitness, OptimalDelay, TimingArray=PathFitnessEvaluator(Child)
			FitnessArray[CIndex]=Fitness
			OptimalDelayArray[CIndex]=OptimalDelay
			TimingArrayMatrix[CIndex]=TimingArray
```

7.2.3.4.4 Parallel Mutation Algorithm

Mutation is a procedure where a randomly chosen chromosome is changed every generation or iteration of the GA. This procedure increases the quality of the solution converged at by the GA and prevents it from converging at a local optimum, increasing the chance of obtaining a globally optimal solution. The mutation strategy used in the GA is straightforward. It involves choosing a random index in the `SectorPath` or chromosome called `MutationPoint` and starting a random DFS from that sector at that index to the `ArrSec`. We have already detailed the random DFS procedure in `Algorithm 1` where we generated the initial population. The parallelization strategy followed is straightforward; we take `M` as an input which is the number of chromosomes to be mutated per generation. We spawn M threads, and each thread mutates a randomly chosen chromosome

```
Algorithm 5
	Parallel Mutation Algorithm
Input
	Population, FitnessArray, arrSec, Graph
Output
```

```
	-
Code
	parallel foreach ThreadID ∈ {0 … M}
		ChromosomeIndex=random.choice(Population.length())
		Chromosome=Population[ChromosomeIndex]
		MutationPoint=random.choice([0…Chromosome.length()])
		OriginalPath=Chromosome[0…MutationPoint-1]
		DepSec=Chromosome[MutationPoint]
		RandomPath=randomDFS(DepSec,arrSec,Graph)
		MutatedChromosome=OriginalPath+RandomPath
		Population[ChromosomeIndex]=MutatedChromosome
		Fitness, OptimalDelay, TimingArray=PathFitnessEvaluator(Chromosome)
		FitnessArray[Chromosome.index]=Fitness
		OptimalDelayArray[Chromosome.index]=OptimalDelay
		TimingArrayMatrix[Chromosome.index]=TimingArray
```

7.2.3.4.5 Parallel Repair Function

The chromosomes or paths must be free from cycles to maintain a good fitness value, and the previous steps, Crossover and Mutation, would add cycles to the paths generated. To eliminate these cycles, we use a repair function that fixes the cycles in the chromosomes. The repair function maintains a dictionary storing the latest position of each sector, and any repeated sector (resulting in a cycle) is eliminated this way. The parallelization strategy involves generating `3P/2` threads, one for each chromosome in the `Population`, and each thread will eliminate any possible cycle in its allocated chromosome.

```
Algorithm 6
	Parallel Repair Algorithm
Input
	Population, FitnessArray
Output
	-
Code
	parallel foreach ThreadID ∈ {0 … 3P/2}
		Chromosome=Population[ThreadID]
		Dictionary=Hashtable()
		RepairedPath=[]
		for i in [0, Chromosome. length()]:
			Dictionary[Chromosome[i]]=i
		index=0
		while(index< Chromosome.length()):
			RepairedPath.append(Chromosome[index])
```

```
                index=Dictionary(Chromosome[index])+1
        Population[ThreadID]=RepairedPath
        Fitness, OptimalDelay, TimingArray=PathFitnessEvaluator(RepairedPath)
        FitnessArray[ThreadID]=Fitness
        OptimalDelayArray[ThreadID]=OptimalDelay
        TimingArrayMatrix[ThreadID]=TimingArray
```

7.2.3.4.6 Parallel Elimination Algorithm

The Elimination procedure involves marking the chromosomes with the lowest Fitness Values to be replaced with the offspring in the next iteration of the GA. We use a parallel sort algorithm, available as a CUDA thrust module, to find the `P/2` chromosomes with the lowest fitness and then mark their indices in the `ReplacementPool`. As mentioned earlier, the `ReplacementPool` contains indices of the `Population` array to be overwritten by the offspring of the crossover algorithm. By eliminating/overwriting the worst chromosomes from the `Population`, we could observe that the GA converges in far fewer generations, drastically improving the algorithm's performance.

The parallelization strategy is as follows, we spawn `3P/2` threads, and we generate an array of objects (`SortArray`) like so `{i, FitnessArray[i]}`. Then we use the CUDA Thrust sort to sort the array of objects using the `FitnessArray[i]` as the key in descending order. So, the last `P/2` objects will have the lowest Fitness value. We then set `ReplacementPool` as the indices where these `P/2` lowest fitness chromosomes occurred.

```
Algorithm 7

        Parallel Elimination Algorithm
Input
        FitnessArray, ReplacementPool
Output
        -
Code
        ObjectArray=[]
        parallel foreach ThreadID ∈ {0 … 3P/2}
                ObjectArray[ThreadID]={
                                index =threadID,
                                fitness=FitnessArray[ThreadID]}
        SortedObjectArray=thrust::sort(ObjectArray with key as fitnessArray)
        parallel foreach ThreadID ∈ {0 … P/2}
                ReplacementPool[ThreadID]= SortedObjectArray[P+ThreadID].index
```

7.2.3.4.7 Convergence Criteria

We state that the GA converges if the maximum fitness of any chromosome in the Population remains the same for 20 generations. The test is relatively simple; we store the maximum value of `FitnessArray` as a variable (`ConvergenceFactor`) outside the loop after every generation and count the number of times this variable takes the same value. If `ConvergenceFactor` remains the same for twenty consecutive generations, we conclude that the algorithm has converged to a solution. We support this argument by stating that if the population's best solution has not improved for twenty generations, then an optimum is reached, and further generations will not improve it.

7.2.3.5 The Genetic Algorithm

The above procedures must be executed in a loop until the convergence criterion is satisfied. For every input flight, the GA loop must iterate until convergence, upon which we select the highest `Fitness` chromosome as the optimal solution. For that flight, we return this chromosome and its corresponding `OptimumDelay` as the `OutputPath` and `OutputDelay`, respectively. We also update the `TrafficMatrix` and `AirportMatrix` to reflect the `OutputPath` for that flight. Hence, the parallel genetic algorithm is executed for every flight in the sequence the user enters as an input.

The overall algorithm in the deployed application is detailed below. We use `P=5000` and `M=5` in our model. We depict the CUDA kernel calls below, with the kernel launch parameters being shown in `<<<…>>>`.

```
Algorithm 8
        Parallel Genetic Algorithm
Input
        Flight Schedule with N flights, Airport Information
Output
        Flight Path, OptimumDelay ∀ N Flights
Code
        for FlightID in [0, N-1]:
                Convergence=False
                EP=3P/2
                S=EP/2
                C=S/2
                R=P/2
                Population=[]
                FitnessArray=[]
                OptimumDelay=[]
                ZipPopulation=Object{Population,FitnessArray,OptimumDelay}
                ParallelInitialPopulation<<<(P/32)+1,32>>>(ZipPopulation)
                ReplacementPool=[P…EP-1]
                SelectionPool=[0…P-1]
                CrossoverPool=[0…S-1]
```

```
while(not Convergence):
        SelectedArray=[]
        SelectionPool=ArrayShuffle([0…EP-1]-ReplacementPool)
        CrossoverPool=ArrayShuffle(CrossoverPool)
        ParallelSelection<<<(S/32)+1,S>>>(ZipPopulation,SelectedArray,SelectionPool)
        ParallelCrossover<<<(C/32)+1,C>>>(ZipPopulation,SelectedArray,ReplacementPool,CrossoverPool)
        ParallelMutation<<<(M/32)+1,M>>>(ZipPopulation)
        ParallelRepair<<<(EP/32)+1,EP>>>(ZipPopulation)
        ParallelElimination<<<(R/32)+1,R>>>(ZipPopulation,ReplacementPool)
        Synchronize();
        Convergence=ConvergenceCriteria()
BestChromosome=indexOf(max(FitnessArray))
OutputPath[FlightID]=Population[BestChromosome]
OutputOptimumDelays[FlightID]=OptimumDelayArray[BestChromosome]
UpdateTrafficMatrix(OutputPath[FlightID])
UpdateAirportMatrix(OutputPath[FlightID])
```

7.2.3.6 Results

We output the `TrafficMatrix, AirportMatrix,` and the `SectorPaths` allocated to each input flight. Then, we use these values in the Evaluation module and the Simulator module, which we detail in the following sections.

7.2.4 Website

We develop a web application that allows users to enter a flight schedule and obtain the flight paths, optimal delays, and aerial times. In addition, we will allow the user to seamlessly execute the CUDA GA code and the Simulator module with the hit of a button. First, the user enters the flight's departure and arrival airports, scheduled departure time, and the cruise speed with which the aircraft

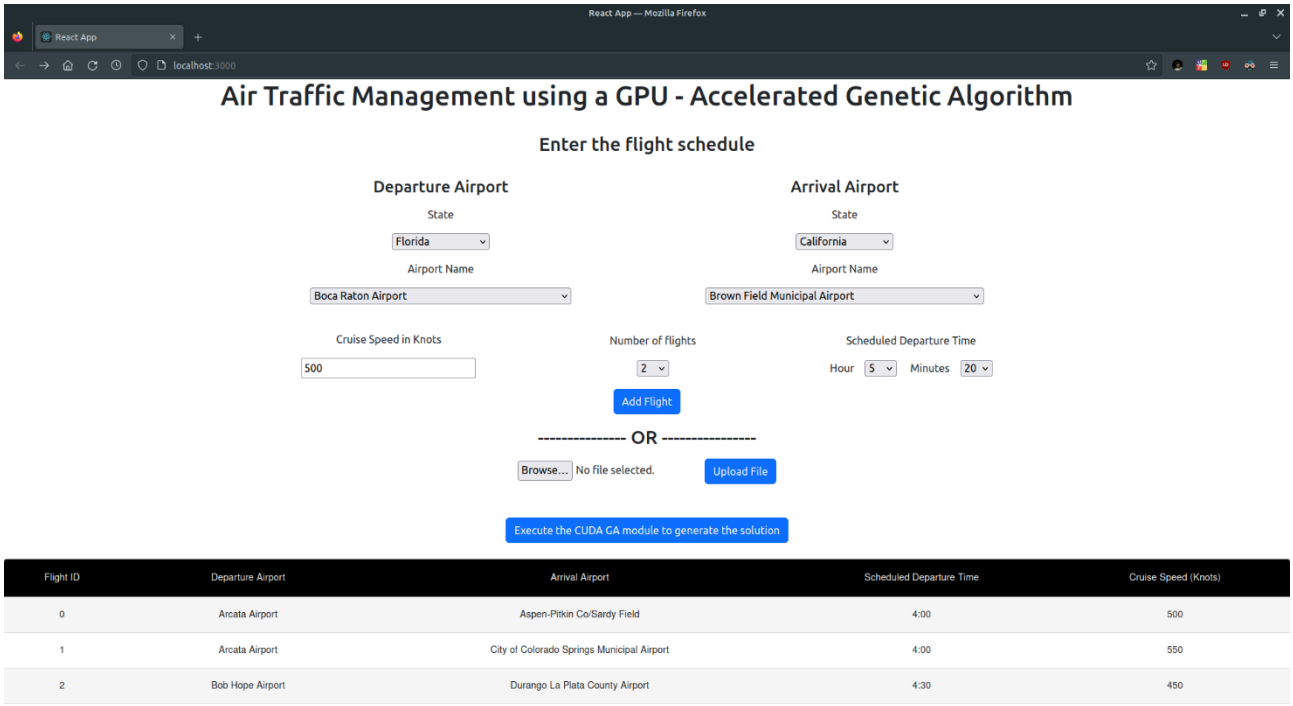
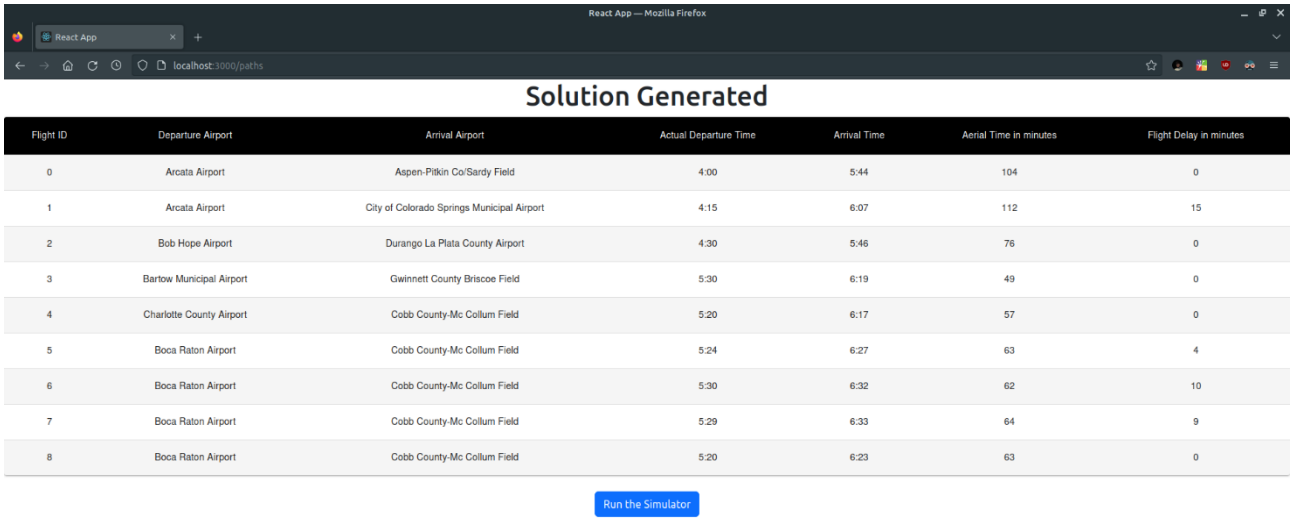


Figure 7.20 The Landing Page

will fly using the interactive frontend developed in ReactJS. The user can also upload a file with the flight schedule instead of entering the data manually. A dynamic table will reflect schedule input by the user and will be constantly updated whenever the user input is modified. We depict the landing page in Figure 7.20

Then, the user clicks a button, and the input is converted to a format the CUDA code understands and is saved as a text file. This file is one of the inputs to the CUDA code called InputFromFrontend.txt. The backend then executes the application and waits asynchronously until the application has finished generating the solution. The backend is written using NodeJS and ExpressJS for effective API Development.

The GA is executed N times for N flights input by the user. After this process is complete with the relevant outputs obtained, the GA creates a file OutputoFrontend.txt which the backend reads and redirects the user to another page with a table showing the flight delay, flight time, and its actual departure time. In addition, the user can run the Simulator on this page, which simulates all the paths generated by the GA by triggering an API that runs a Python script in the backend and displays the simulator.



Flight ID	Departure Airport	Arrival Airport	Actual Departure Time	Arrival Time	Aerial Time in minutes	Flight Delay in minutes
0	Arcata Airport	Aspen-Pitkin Co/Sardy Field	4:00	5:44	104	0
1	Arcata Airport	City of Colorado Springs Municipal Airport	4:15	6:07	112	15
2	Bob Hope Airport	Durango La Plata County Airport	4:30	5:46	76	0
3	Bartow Municipal Airport	Gwinnett County Briscoe Field	5:30	6:19	49	0
4	Charlotte County Airport	Cobb County-Mc Collum Field	5:20	6:17	57	0
5	Boca Raton Airport	Cobb County-Mc Collum Field	5:24	6:27	63	4
6	Boca Raton Airport	Cobb County-Mc Collum Field	5:30	6:32	62	10
7	Boca Raton Airport	Cobb County-Mc Collum Field	5:29	6:33	64	9
8	Boca Raton Airport	Cobb County-Mc Collum Field	5:20	6:23	63	0

Figure 7.21 Page that Displays the Solution

7.2.5 Simulator

The simulator is built from scratch using Python, using matplotlib. We read the `SectorPaths` from the OutputToFrontent.txt file and use the figure shown in Figure 7.8 as the background upon which we show the aeroplanes moving with time. We first serialize the sectorized USA map (shown in Figure 7.8) as a matplotlib figure on the disk using the `pickle` persistent storage mechanism as Simulator.pkl. Whenever the simulator is run, this pickle file is loaded and deserialized. Next, we use `distinctipy`, a library to get N visually distinct colours, and we map each colour to one of the N flight paths being simulated as a one-to-one mapping.

For every `SectorPath` returned by the GA, we obtain its corresponding Path Length(`L`) and its sequence of Connecting Points. Next, we use the `shapely` library's `LineString function` to compact this sequence into one object, `PointsSequence`. We then convert the `CruiseSpeed` of the flight to the Meter per Minute(m/min) unit, which gives us the distance the flight can travel in one minute. Finally, using the `NumPy` library's `arange` method, we divide `L` into a sequence of distances marked by the minute at which that distance is traversed (`SplitDistances = np.arange(0, L, CruiseSpeed))`.

With the `PointsSequence` and the `SplitDistances` for a `SectorPath`, we can obtain the exact 2D coordinates at which the aircraft will be present, given any minute during its flight. We get this information using the `shapely` library's `interpolate` function as `MinutePoints` defined as `[PointsSequence.interpolate(D) for D in SplitDistances] + [PointsSequence.boundary.geoms[1]]`.

Hence for each `SectorPath`, we obtain the corresponding `MinutePoints` and put them in an array `MinutePointsArray`. We now create an array called `TimeArray` with 2880 rows, each corresponding to the respective minute in two days. Then, using the `MinutePointsArray`, we populate the array by mapping every minute to its corresponding set of points.

We then use matplotlib scatterplot to plot all the airports involved in the flight schedule input by the user on the figure, and we show this in Figure 7.22.

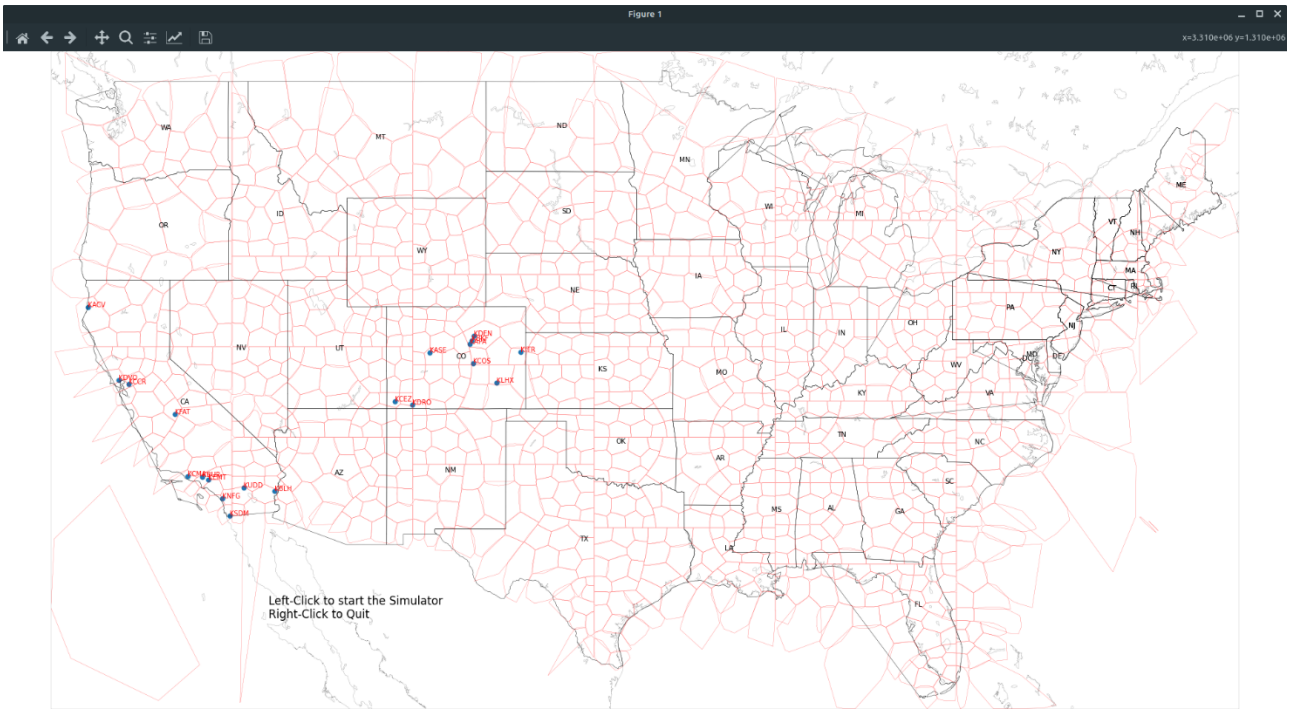


Figure 7.22 Initial Simulator State

We then allow the user to start the Simulator by clicking the left mouse button, upon which we iterate through all the rows of `TimeArray` in a for loop, plotting the points in the `ith` iteration and removing the points plotted in the `(i-1)th` iteration. We color each point plotted with the flight to color mapping obtained earlier. We also display a clock in the right-hand corner that displays the time in 24hr format. We show the progression of the simulator with time in Figures 7.23 and 7.24. We visually observe the dispersion of air traffic in the two figures, with each flight represented with a color. As multiple aircraft approach the same airport, we observe that they form a queue and land after each other, akin to how it happens in real life.

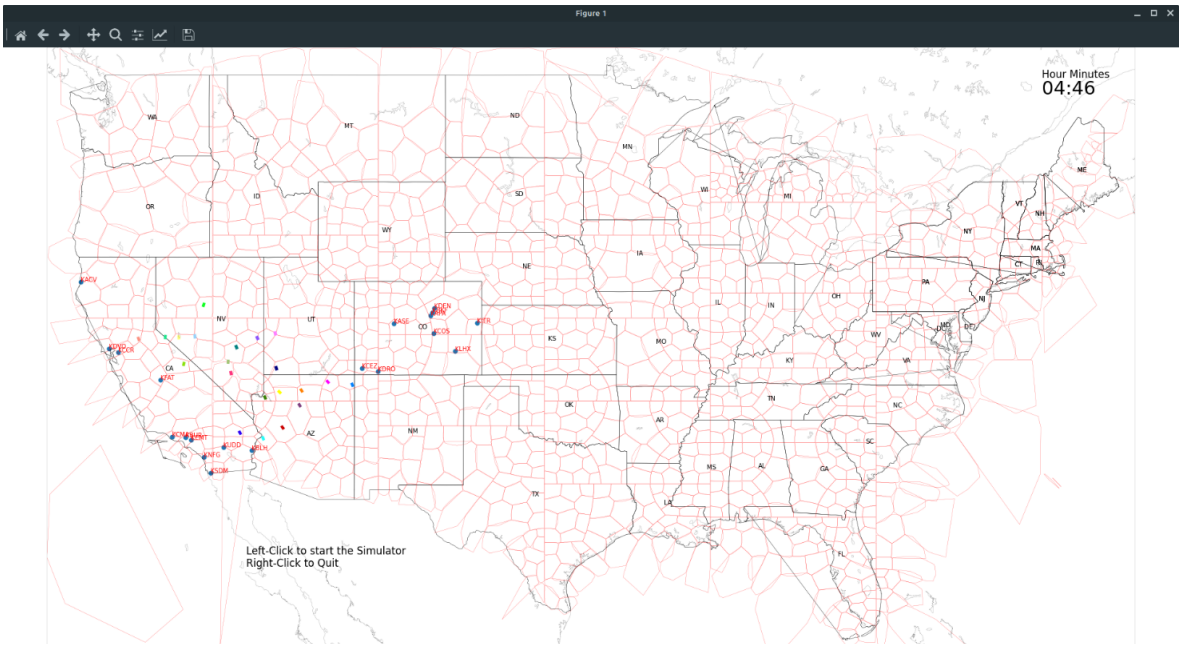


Figure 7.23 Simulator at 4:46

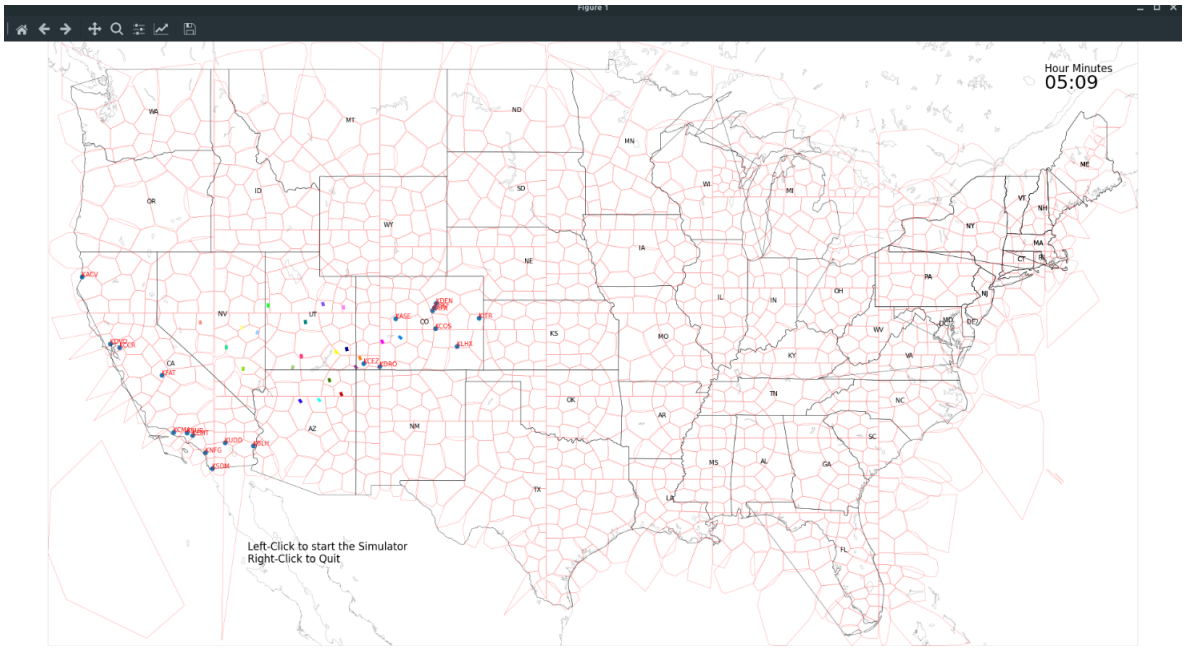


Figure 7.24 Simulator at 5:09

Chapter 8

RESULTS

To evaluate how well our model performs compared to the existing air traffic management system used by the FAA, we compare three different metrics, as stated in Chapter 2. We define these three metrics here and briefly overview the evaluation methodology followed. The main goal of the model is to generate a flight path for an aircraft that satisfies the following criteria

- If an airplane follows this path, it should encounter the least number of aircraft in its vicinity throughout the time it spends traversing the path. We measure this quantity using two terms, `SectorOccupancy` and `SectorDensity`.
- The departure and arrival airport must have at least one runway available for the aircraft to use at the time of departure or arrival, provided that any aircraft blocks a runway for a whole minute if it is using it. We keep this as a constraint in the Fitness Evaluator.
- The flight delays incurred due to traffic must be minimal. We measure this using the quantity using the term `GroundDelay`.
- The flight time, which is the time spent in the air by the aircraft, must be minimal. We measure this using the quantity using the term `AerialTime`.

With these points in mind, we use the ASDI Flight History data table from our dataset to get a list of all the domestic flights that took place in the USA on 19th August 2013. We found out that 23,893 flights took place that day, and we obtained the flight schedule for that day which had each flight's scheduled departure time and departure and arrival airports. We upload this flight schedule to the website and run the algorithm on a machine with the RTX 3050 GPU. We modify the application to output the `TrafficMatrix` and `AirportMatrix` used internally, along with the `SectorPath`, `OptimumDelay`, and `AerialTime` for each of the 23,893 flights. We run the parallel GA 23,893 times, each providing the corresponding flight's output with a total execution time of around 4 hours.

Hence after execution of the GA model, we get the `TrafficMatrix`, `AirportMatrix`, and each flight's `AerialTime` and `OptimumDelay`. We use the ASDI Flight Tracks table from our dataset to obtain the flight paths taken in reality by the 23,893 flights on 19th August 2013 and get the corresponding `RealTrafficMatrix`, `RealAirportMatrix`, and each flight's `RealAerialTime` and `RealOptimumDelay`. With this information, we are ready to evaluate the model's performance. We compare the `AerialTime` with `RealAerialTime`, `OptimumDelay` with `RealOptimumDelay`, and use the Traffic and Airport matrices to measure the `SectorDensity`, and `SectorOccupancy` and check the runway constraint satisfaction.

8.1 Flight Time

We use `RealAerialTime`, which is an array of flight times for all the 23,893 flights that took place on 19th August 2013 in reality, and `AerialTime`, which is an array of flight times for the same flights provided by our GA model. We subtract each flight's flight time from `RealAerialTime` and `AerialTime` and store it in an array `FlightTimeDiff` as so:

```
for flight in 2013-08-19:
        FlightTimeDiff.append(RealAerialTime[flight]-AerialTime[flight])
```

We plot a histogram of FlightTimeDiff, and we show this in Figure 8.1. We measure the average value of `FlightTimeDiff` and depict it in Table 8.1. We justify the variance in flight times by stating that we are assuming a constant cruise speed and not accounting for the acceleration and deceleration the flight undergoes while ascending and descending, respectively. We also observe that the mean difference is 6.52 minutes, which means that, on average, our model reduces the flight time of a flight by approximately seven minutes.

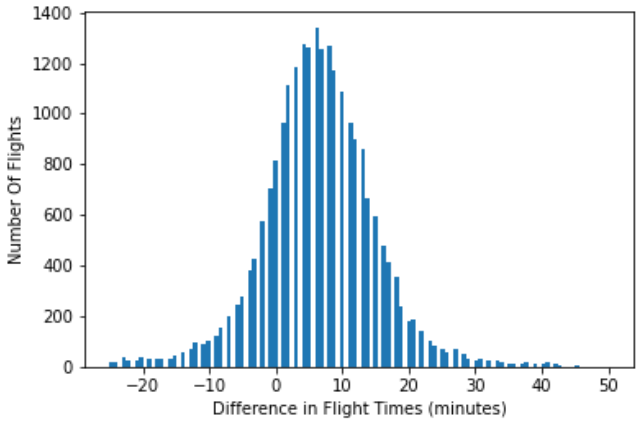


Figure 8.1 Real Flight Time - GA Flight Time

Average of `FlightTimeDiff`	*6.52 minutes*

Table 8.1 Reduction in Flight Times

8.2 Flight Delays

We use `RealOptimumDelay`, which is the set of delays that each of the 23,893 flights that took place on 19th August 2013 experienced in reality, and `OptimumDelay`, which is the set of delays the GA model imparts to the same set of flights. We show that we reduce the average flight delay by five minutes in our model in Table 8.2. We also plot two histograms, one for `RealOptimumDelay` and one for `OptimumDelay`, and display them in Figure 8.2. We remove the flights with greater than two-hour delay from `RealOptimumDelay`, citing them as outliers, and remove the same from `OptimumDelay`. With this, we show that our model not only imparts minimal delay to flights for managing air traffic but also ensures that the flight delays are reduced by five minutes from the delays imparted to the flights in reality.

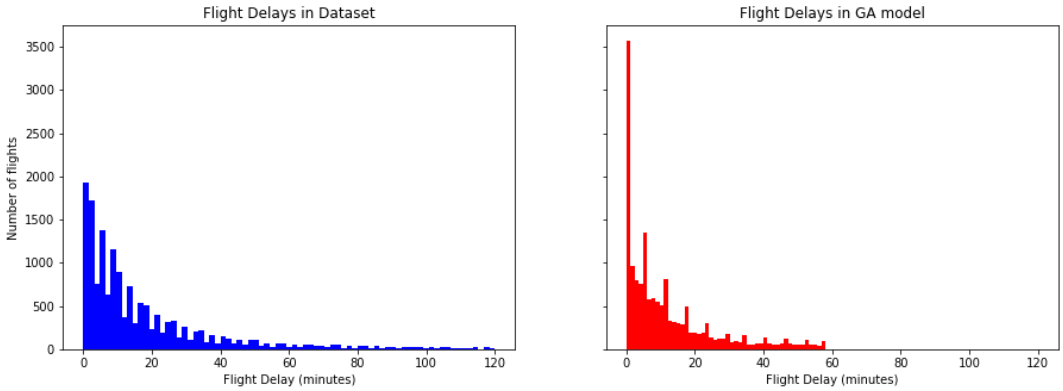


Figure 8.2 Flight Delays

	Dataset	*GA Model*	*Reduction*	*Percentage Reduction*
Average Flight Delay	18.26 min	13.14 min	5.12 min	28.03%

Table 8.2 Reduction in Flight Delays

8.3 Air Traffic

We use `RealTrafficMatrix`, which is the `TrafficMatrix` calculated using the flight paths taken in reality by the 23,893 flights that took place on 19th August 2013 and the `TrafficMatrix` returned by the GA, to evaluate the reduction in air traffic.

We define two methods for evaluating air traffic, `SectorDensity` and `SectorOccupancy`, which we explain here. `SectorDensity` is the primary measure of air traffic and is the maximum number of aircraft present in a sector throughout the day. We measure this quantity by taking the maximum value of each row of the `TrafficMatrix`. We obtain an array of 1250 values, where `SectorDensity[i]` is the maximum number of aircraft present in sector `i` throughout the test date of 19th August 2013. Using `RealTrafficMatrix`, we obtain `RealSectorDensity` and compare it with `GASectorDensity`, which we get using the `TrafficMatrix` output by the GA. We show a histogram plotting `RealSectorDensity` and `GASectorDensity` in Figure 8.3. We observe a reduction in the number of sectors with high `SectorDensity` in the GA output, proving that the traffic hotspots are reduced in our model and that the air traffic is distributed. We take the average of `RealSectorDensity` and `GASectorDensity` and show them in Table 8.3. We observe that, on average, the `SectorDensity` has reduced by 18% compared to the dataset. With that, we conclude that our model reduces Air Traffic by 18% using the measure of `SectorDensity`. We also observe that, compared to `RealSectorDensity`, we reduce the `GASectorDensity` in 55.6% of the sectors and in 77.28% of the sectors, we either reduce or equal `RealSectorDensity`.

Figures 8.4 and 8.5 show two traffic heatmaps comparing the `SectorDensities` in the Dataset and the GA Solution, where we assign a white-to-black colour gradient to a sector based on its `SectorDensity`.

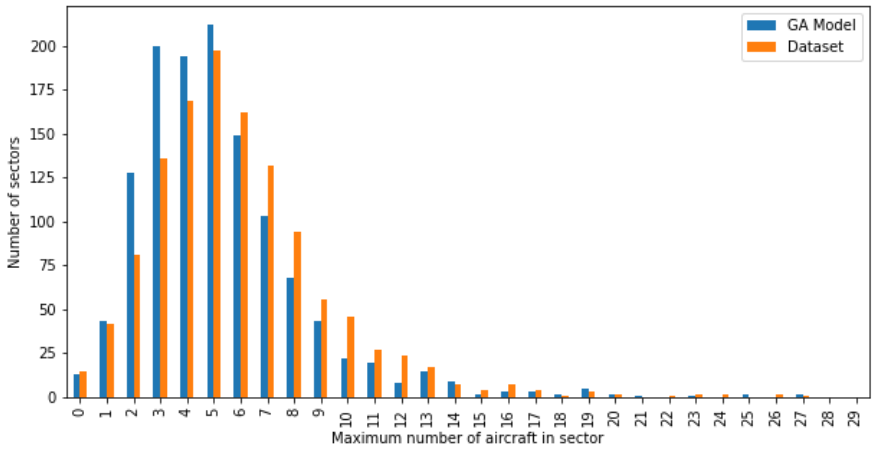


Figure 8.3 Sector Densities from dataset and GA model

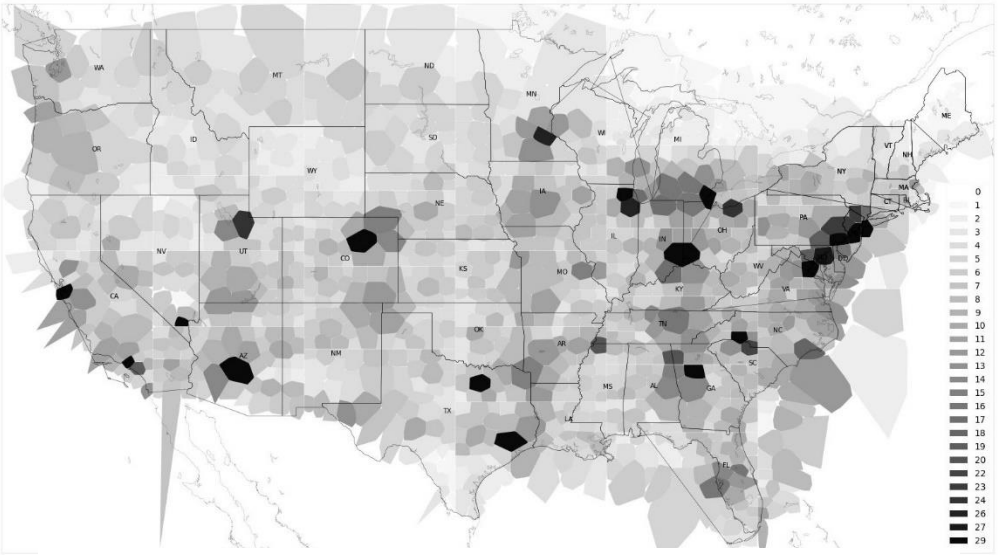


Figure 8.4 Traffic Heatmap of Dataset

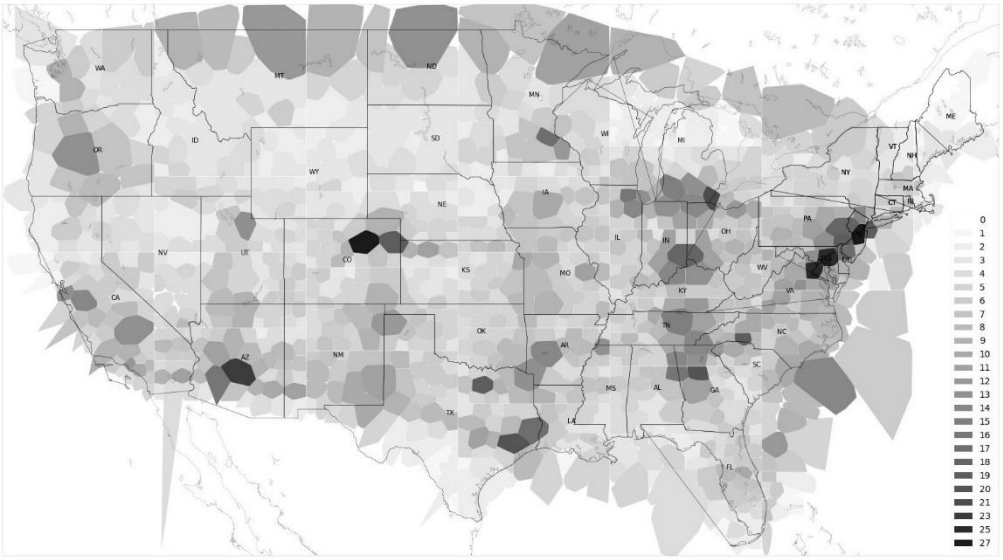


Figure 8.5 Traffic Heatmap of GA Solution

`SectorOccupancies` is the other method of measuring air traffic wherein we take the average of each row of `TrafficMatrix` instead of max. Hence we obtain an array of size 1250, where `SectorOccupancies[i]` is the average number of aircraft present in the i^{th} sector throughout the test day.

Hence, using the `RealTrafficMatrix`, we obtain `RealSectorOccupancies`, and with the `TrafficMatrix` output by the GA, we get the `GASectorOccupancies`. With this measurement, we find that the average SectorOccupancy across all the sectors has reduced by 9%. We hence conclude that our model reduces Air Traffic by 9% using the measure of `SectorOccupancy`. We observe that in 58.32% of sectors, we reduce the `SectorOccupancy` and in 61.04% of the sectors, we reduce or equal the `SectorOccupancy`, compared to `RealSectorOccupancies`.

	Dataset	*GA Model*	*Reduction*	*Percentage Reduction*
Average Sector Density	6.42	5.26	1.16	18.06%
Average Sector Occupancy	2.05	1.87	0.18	8.78%

Table 8.3a Reduction in Air Traffic

	Sector Density is Reduced	*Sector Density is Reduced or Equal*	*Sector Occupancy is Reduced*	*Sector Occupancy is Reduced or Equal*
Percentage of all sectors	55.6%	77.28%	58.32%	61.04%

Table 8.3b Percentage of sectors where Air Traffic is Reduced

8.4 Airport Runway Constraint

We state that, for every airport, the maximum number of aircraft it can allow to depart and arrive concurrently is equal to the number of runways it has because each departing or arriving aircraft is allocated or mapped to a runway. It blocks that runway for a whole minute. When a runway is blocked, no other aircraft can use it and hence must utilize some additional runway or delay itself until a runway is freed up. The runway blocking constraint must be satisfied for all flights to and from an airport, and we prove that our model always satisfies this constraint in this section. We use the `AirportMatrix` output by our GA model to check for the satisfaction of these constraints. `AirportMatrix[i][j]` represents the number of runways used concurrently in `Airport i` in the j^{th}

minute of the day. Hence we check the condition `max(AirportMatrix[i]) < NumberOfRunways[i]` for all the Airports involved in the test day 2013-08-19. We plot the percentage of airports where the above constraint is obeyed in Table 8.4. We observe that is constraint is followed in 100% of the airports. We also plot a pie chart in Figure 8.5 that shows the relationship between the percentage of times in a day x number of runways were used concurrently for the busiest airport in the USA, Hartsfield-Jackson Atlanta International Airport (KATL), which has five total runways available. We see that 262 times (25.4% of the time) one runway was used and 346 (33.5% of the time) two runways were used concurrently, meaning a flight was mapped to the second runway because it is observed that the first runway was blocked or occupied by some other aircraft. We make the same argument to state that 280 times in a day (27.1% of the time), both runways 1 and 2 were blocked; hence, the third runway was used. We observe that six runways are not used concurrently in the pie chart since KATL has only five runways, thus proving that our constraint holds. If all five runways were blocked, then the aircraft was delayed until one of the runways was freed.

Number Of Airports involved	*Number of Airports where the constraint is satisfied*	*Number of Airports where the constraint is not satisfied*	*Percentage of Airports where the constraint is satisfied*
619	619	0	100%

Table 8.4 Constraint Satisfaction

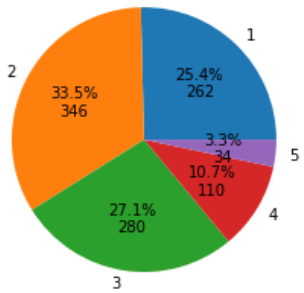


Figure 8.4

Number of runways used concurrently in KATL

REFERENCES

[1] Christian Kiss-Tóth; Gabor Takács "A dynamic programming approach for 4D flight route optimization", 2014 IEEE International Conference on Big Data (Big Data)

[2] Sudarshan Vaidhun; Zhishan Guo; Jiang Bian; Haoyi Xiong; Sajal K. Das, "Priority-based Multi-Flight Path Planning with Uncertain Sector Capacities", 2020 International Workshop on Advanced Computational Intelligence (IWACI)

[3] Han Wen; Hui Li; Zhuang Wang; Xianle Hou; Kangxin He, "Application of DDPG-based Collision Avoidance Algorithm in Air Traffic Control", 2019 International Symposium on Computational Intelligence and Design, ISCID

[4] TongHe; IrajMantegh; Long Chen; Charles Vidal; Wenfang Xie, "Flight Path Planning for Dynamic, Multi-Vehicle Environment ", 2020 International Conference on Unmanned Aircraft Systems (ICUAS)

[5] Hang Zhou, Xiao-Bing Hu," A Ripple Spreading Algorithm for Free-Flight Route Optimization in Dynamical Airspace", 2020 IEEE Symposium Series on Computational Intelligence (SSCI)

[6] Xiao-Bing Hu, Shu-Fan Wu, Ju Jiang, "Online free-flight path optimization based on improved genetic algorithms", Engineering Applications of Artificial Intelligence Volume 17, Issue 8, December 2004, Pages 897-907

[7] Chang Wen Zhenga, MingyueDingb, ChengpingZhoub, Lei Lia, "Co-evolving and cooperating path planner for multiple unmanned air vehicles", Engineering Applications of Artificial Intelligence Volume 17, Issue 8, December 2004, Pages 887-896

[8] Yang Liu, Xuejun Zhang, Yu Zhang, Xiangmin Guan, "Collision-free 4D path planning for multiple UAVs based on spatial refined voting mechanism and PSO approach", Chinese Journal of Aeronautics Volume 32, Issue 6, June 2019, Pages 1504-1519

APPENDIX – A

DEFINITIONS, ACRONYMS, AND ABBREVIATIONS

1. ATC – Air Traffic Controller
2. GA – Genetic Algorithms
3. UAV – Unmanned Aerial Vehicles
4. FF – Free flight
5. PSO – Particle Swarm Optimization
6. OD – Origin Destination
7. ASDI – Aircraft Situation Display to Industry
8. GE – General Electric
9. CEPO – Co Evolutionary path optimization
10. OLRO – Online Re-optimization
11. TAS – True Air Speed
12. MDP – Markov Decision Process
13. UAM – Urban Aerial Mobility
14. APF – Artificial Potential Field
15. DDPG – Deep Deterministic Policy Gradient
16. DPG – Deterministic Policy Gradient
17. DQN – Deep Q-Network
18. RL – Reinforcement Learning
19. ARTCC – Air Route Traffic Control Centres
20. FAA – Federal Aviation Administration
21. ETMS – Enhanced Traffic Management System

Air Traffic Management using a GPU-Accelerated Genetic Algorithm

ORIGINALITY REPORT

8%	4%	6%	3%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Submitted to PES University Student Paper	2%
2	Submitted to University of Brighton Student Paper	1%
3	Hang Zhou, Xiao-Bing Hu. "A Ripple Spreading Algorithm for Free-Flight Route Optimization in Dynamical Airspace", 2020 IEEE Symposium Series on Computational Intelligence (SSCI), 2020 Publication	1%
4	www.ece.ucf.edu Internet Source	1%
5	pdxscholar.library.pdx.edu Internet Source	1%
6	Tong He, Iraj Mantegh, Long Chen, Charles Vidal, Wenfang Xie. "UAS Flight Path Planning for Dynamic, Multi-Vehicle Environment", 2020 International Conference on Unmanned Aircraft Systems (ICUAS), 2020 Publication	1%

7	www.uasconferences.com Internet Source	<1 %
8	Hu, X.-B.. "On-line free-flight path optimization based on improved genetic algorithms", Engineering Applications of Artificial Intelligence, 200412 Publication	<1 %
9	www.sze.hu Internet Source	<1 %
10	Yang LIU, Xuejun ZHANG, Yu ZHANG, Xiangmin GUAN. "Collision free 4D path planning for multiple UAVs based on spatial refined voting mechanism and PSO approach", Chinese Journal of Aeronautics, 2019 Publication	<1 %
11	Han Wen, Hui Li, Zhuang Wang, Xianle Hou, Kangxin He. "Application of DDPG-based Collision Avoidance Algorithm in Air Traffic Control", 2019 12th International Symposium on Computational Intelligence and Design (ISCID), 2019 Publication	<1 %
12	docplayer.net Internet Source	<1 %