Aircraft Traffic Control with Reinforcement Learning

*A project for the course CS 238: Decision Making Under Uncertainty

Aamir Rasheed
Computer Science
Stanford University
Stanford, CA
aamirar@stanford.edu

Ashar Alam
Mechanical Engineering
Stanford University
Stanford, CA
ashar1@stanford.edu

John Melloni
Computer Science
Stanford University
Stanford, CA
jmelloni@stanford.edu

Ron Domingo
Electrical Engineering
Stanford University
Stanford, CA
rdomingo@stanford.edu

***Abstract*—A critical component of air traffic control (ATC) is the task of directing airplanes to takeoff and land at airports without collisions. It is a complex task involving a sequence of complex decisions. We simulate a 2D version of ATC and describe an approach to train an agent to route planes without collissions automatically. We frame the problem as a Markov decision process and utilize the SARSA reinforcement learning algorithm to manipulate the trajectories of the planes to reach their desired destinations without collisions. Our environment features uncertainties in the form of variability of spawn locations, number of planes, initial heading direction of the planes, and unknown destination locations.**

***Index Terms*—Air Traffic Control, Markov Decision Processes, Reinforcement Learning, Decision Making Under Uncertainty**

I. Introduction

Air traffic control (ATC) has been an integral part of the aviation industry since the early 1920s. Initially, it was used for military aircraft, but with the advent of passenger and cargo planes, air traffic increased over time and ATC evolved into its current form. ATC not only coordinates take-offs and landings of airplanes, but also maintains an oversight on planes throughout their journeys.

With increasing air traffic and instances of mid-air collisions, the FAA (Federal Aviation Administration) developed a mid-air collision avoidance system known as TCAS (Traffic alert and Collision Avoidance System) in 1981. A new system called ACAS-Xu (Airborne Collision Avoidance System) is currently being developed which utilizes advancements in computer science and dynamic programming to generate alerts [1].

Our general approach is to send planes on straight-line trajectories towards their destinations, with an ACAS-like, SARSA-based agent that takes over when two planes are in danger of collision.

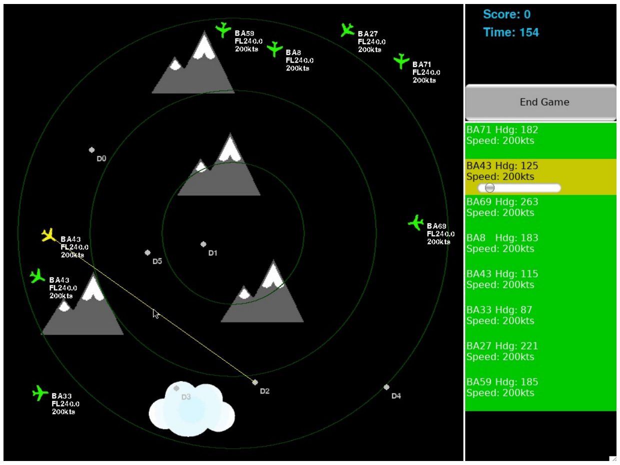
II. Related Work

Reinforcement learning and dynamic programming have been utilized extensively in solving the problems of ATC. One such issue with Markov decision processes (MDPs) and partially observable Markov decision processes (POMDPs) is the size of the state space used for collision avoidance. In Policy Compression for Aircraft Collision Avoidance Systems, Julian Kyle et al, discuss several avenues they explored for reducing the state space [1]. To illustrate this point, here is an introduction of a novel system: ”A variant [of a collision avoidance system] for unmanned aircraft, ACAS Xu, uses dynamic programming to determine horizontal or vertical resolution advisories in order to avoid collisions while minimizing disruptive alerts.” By keeping track of 7 discretized sensor measurements, including distance, angle, and velocity of ownship and intrudership, the ACAS Xu logic is left with 120 million different discrete states [1]. Even by limiting the action space to only 5 actions, that results in 600 million state action pairs to be used in the look-up table. Thanks to the work of Professor Kochenderfer and others, they were able to downsample, compress identical states, and even store the values in IEEE half-precision format with no loss in performance [2]. Kyle and others introduced Origami Compression and Deep Neural Network Compression algorithms to further reduce the magnitude of the look-up table. Origami Compression involves decomposing the massive look-up table into a ”set of lower-dimensional tables” that can then be reduced even further via similarity metrics and symmetry exploitation [1]. Deep Neural Network Compression approach learns a ”complex non-linear function approximation of the table.” With these new algorithms, they were able to give a more accurate compression than current approaches, and even increase runtime speed in the case of Origami Compression [1].

III. Environment

We decided upon using PyGame for our visualization environment. PyGame is a set of Python modules used for programming video games [3]. More specifically, we used an open source air traffic control game built on top of PyGame called *python-air-traffic-control* [4].

Fig.1 is an example of a typical gameplay where multiple planes try to reach multiple destinations while avoiding ob-



Score: 0
Time: 154

BA71 Hdg: 182
Speed: 200kts

BA43 Hdg: 125
Speed: 200kts

BA69 Hdg: 263
Speed: 200kts

BA8 Hdg: 183
Speed: 200kts

BA43 Hdg: 115
Speed: 200kts

BA33 Hdg: 87
Speed: 200kts

BA27 Hdg: 221
Speed: 200kts

BA59 Hdg: 185
Speed: 200kts

Fig. 1. Our PyGame Visualization Environment.

stacles. Each plane begins with a straight line trajectory to its destination airport but has the ability for waypoints to be added along that trajectory to ensure that no planes collide into other planes or static objects. The goal of the game is to ensure that all planes can reach their destination without any collisions. Normally, the game is played with a human user manually clicking individual planes and inputting their next waypoints. Therefore, many modifications were made to the game to enable our autonomous agent to have full control and be able to learn from previous runs. The modifications made can be split into two categories: (1) Game Modifications and (2) Interface Modifications.

A. Game Modifications

First, all aspects of the original game that were outside of the actual gameplay, such as the start menu, were stripped to allow for continuously training over multiple episodes without interruption. The game was also modified so that it has to be iteratively stepped through in order to play. This allowed us to generate and execute our desired actions at every timestep of the game. Custom functions were written within the game code so that, with each timestep, the game returns the current list of active aircraft, the rewards for each aircraft, and the list of aircraft pairs in potential collision courses - all of which constitute the information our MDP needs to make decisions about the game state.

B. Interface Modifications

We created an interface that wraps around the game code and allows us to programmatically control the game, train our model, and actively make decisions given the game state. Our interface enables us to modify the default game configurations that detail the number of planes, destinations, and obstacles per game, as well as the hyperparameters of the SARSA algorithm, such as learning rate, discount factor, and exploration probability. The interface is also the controller that steps through the game, receiving the state information from each game timestep, processing the data, then generating an action based on the trained model. With each timestep, the interface calculates the best action for every active plane and modifies the game plane objects to add any waypoints it deems necessary before initiating a new timestep to continue the game. Given this interface, we could then create an agent to autonomously handle the task of air traffic control.

IV. General Approach

The problem of optimally routing planes without collisions while taking into account uncertainty of new plane spawns can become a complicated and computationally expensive task. To simplify the problem, we have instead chosen to use a two-pronged approach:

1) If there are other planes within a specified radius, deploy a collision-avoidance agent to closest two planes until they are clear of each other, or a plane reaches a destination.
2) Otherwise the path is clear, therefore make a beeline for the destination.

This approach allows us to mimic the TCAS/ACAS system for collision avoidance while keeping the complexity of the problem at a minimum.

The following section describes step 1 of this approach: the agent that deploys to both planes. For this agent, we chose to use SARSA, a model-free reinforcement learning algorithm. The following sections describe this algorithm's structure.

V. MDP Model

A. MDP Architecture

Similar to the ACAS/TCAS system, our task was to generate the optimal policy that a plane could follow to avoid a collision with an intruder aircraft nearby. The MDP formulation for SARSA requires a definition of the state space, action space, and reward model, which we will describe next.

B. State Space

Our state space formulation is inspired by the ACAS-Xu system and measurements real aircraft might receive about other nearby aircraft. We obtain these 'measurements' in real time and update the agent's states accordingly. Table 1 describes our state space. We have discretized the measurements into buckets to reduce the size of our state space and compress the look-up table. The angle to intruder and relative heading of intruder are each 36 buckets of 10 degree intervals, representing the full 360 degrees. The distance to the intruder is measured in pixels, where 50 is the threshold radius for deploying the agent. The full size of our state space is 64,800, which is a fairly compact representation

TABLE I
State Space

State variable	Description	# of elements
d	Distance to intruder	50
ρ	Angle to intruder	36
θ	Relative heading of intruder	36

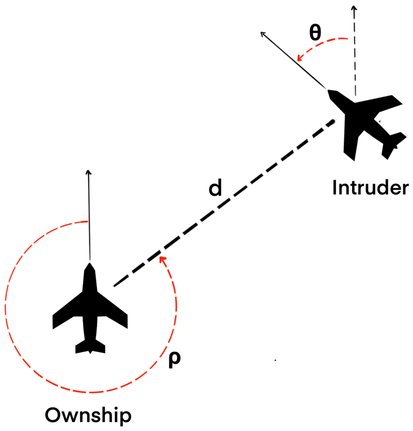


Fig. 2. Visual description of state space (inspired by ACAS-Xu formulation).

compared to the 120 million states needed for ACAS-Xu.

C. Action Space

Our action space consists of going straight (N), taking a medium turn (ML, MR), or taking a hard turn (HL, HR).

To translate this to the pixel space, we utilize the 8 pixels immediately surrounding an aircraft's center pixel. N corresponds to setting the next direction vector of the aircraft to the top center pixel. In a similar fashion, ML and MR correspond to the top left and top right pixels, while HL and HR correspond to the left center and right center pixels. The action itself is set 50 pixels away in the direction of the action's direction vector. A visualization is shown below.

TABLE II
ACTION SPACE

Action	Description
N	Maintain course
HL	Hard left
ML	Medium Left
MR	Medium Right
HR	Hard Right

As was done in [1], we have chosen to limit our action space to 5. This results in a state-action pair look-up table of 324,000 which was tractable for our processing power.

D. Reward Model

The selection of a reward function used in our dynamic programming algorithm is a significant design choice. It allows us to control which information is propagated from past experiences to influence present decisions. Our reward model is dependent on the state variables. Ultimately, we decided to focus on two components, distance and destination.

For distance, we want the negative reward to increase quadratically as intruder planes became closer to our ownship. This means our algorithm would be less likely to choose actions that resulted in a decrease of distance between any two planes. Equation (1) describes the function used. *radius* is the threshold distance of 50 for deploying the agent and *closest_distance* is the distance between any plane and the plane it is closest to. By using the scaling factor in the denominator, it allows us to vary the negative reward from 0 when the planes are 50 pixels away, to 500 when the planes are about to collide.

We also chose to create a positive reward for reaching a destination so that the agent would be motivated to route the plane to a destination in spite of a nearby intruder. To handle this, we created a function called getDistanceToGo() that calculated the distance from the current ship to its destination. Equation (2) describes our reward for approaching a destination. In the below equation, distanceToGo is the result of getDistanceToGo().

$$\text{intruder reward} = \frac{-(radius^2 - closest\_distance^2)}{(radius^2/500)} \tag{1}$$

$$\text{destination reward} = 100 - distanceToGo \tag{2}$$

Both of these rewards are distributed to all agents per timestep in the update function of our learning algorithm. Additionally, to account for planes that have already reached the destination, we propagate the reward of reaching the terminal state back throughout the look-up table.

VI. Implementation

The basis for our implementation primarily comes from Professor Kochenderfer's Decision Making under Uncertainty textbook [5]. Initially, we thought that we should use a model-based learning algorithm as our transitions between states were deterministic. However, in this project, we were focusing on avoiding collisions and could not predict which actions would lead to collisions and which ones would not.

Since we wanted our agent (planes) to learn how to avoid collisions through actual experience rather than through an unknown collision model, we decided on adopting a model-free learning approach using Temporal Difference [5]. This also enabled us to introduce stochastic elements and a large sequence of state-action pairs. Temporal Difference algorithms enable an agent to learn through every action it takes by updating the knowledge of the agent on every time step rather than on every episode (reaching goal/end state). This knowledge update is the update of the estimate of the utility given by:

$$New \leftarrow Old + \alpha(Target - Old) \tag{3}$$

Here the new estimate is proportional to the difference in target utility and the previous estimate. α in the above equation is called the *learning rate* and is representative of the step size applied in the update with values between 0 and 1.

A. Algorithm Used - SARSA

We chose *SARSA* as our reinforcement learning algorithm over Q-learning as SARSA uses the actual action taken at s_{t+1} to update Q values instead of maximizing over all possible actions as done in Q-learning. SARSA is an *on-policy learning algorithm*, meaning it learns the value of the policy being carried out by the agent, including the exploration steps, while the policy continues to be followed.

Algorithm 1: SARSA

```
1  Initialize Q(s,a) for all s $\in$ S and a $\in$ A
2  t $\leftarrow$ 0
3  repeat for each episode
4      $s_0 \leftarrow$ initial state
5      Choose A from S based on policy derived from Q
         ($\epsilon$-greedy)
6      repeat for each step of episode
7          Choose action $a_t$ based on policy derived from
             Q ($\epsilon$-greedy)
8          Observe new state $s_{t+1}$ and reward $r_t$
9          Q($s_t, a_t) \leftarrow Q(s_t, a_t) +$
10                 $\alpha(r_t + \gamma max_a Q(s_{t+1}, a_t) - Q(s_t, a_t))$
11         s $\leftarrow s_{t+1}; a \leftarrow a_{t+1}$
12     until until s reaches goal state;
13 until episodes over;
```

B. SARSA Implementation

We implemented our MDP structure as a *class* in Python 3.7. We will briefly describe how this algorithm helps to train our agents to avoid collision between each other (*the ordered list correspond to line numbers in the above algorithm*):

1) We initialize the Q table as an empty dictionary.
2) The first episode starts alongside the start of the game with multiple planes spawning in random locations heading towards a destination airport (each episode represents one instance of training without collision).
3) We save an offline policy look-up table (Q table) after simulation of every 25 episodes which may be used later.
4) We initialize all unknown states with a 0 vector.
5) We choose an action based on an already trained look-up table saved previously.
6) We run the game in timesteps permitted by the environment.
7) We train our MDP for planes which are within the collision radius and choose an action (turning in a particular direction) using an exploration vs. exploitation strategy.
8) We observe the next state and reward (low reward for maneuvers, which might lead to collision) gained after taking the previous action and ending up in the next state.
9) Based on the previous action, state and reward; we update our Q - values.
10) Finally, we update the state and action of the planes
11) We repeat this until the planes reach their destination (goal state) or until there is a collision.
12) We train the model until the desired number of episodes, and the learned policy can be saved in the form of a Q-table.

C. Simulation Configuration

We have modified our simulation environment such that it can accept various parameters. Thus, we can run our simulation with various configuration parameters to train our agents under various conditions for regressive training (similar to various difficulty levels for games). Following are the parameters, which can be varied during training:

- number of planes
- number of spawnpoints
- number of destinations
- number of obstacles
- load a previously trained q-table
- learning rate
- discount factor
- exploration probability

VII. Results

A. Experiments

To measure the performance of our RL agent, we simulated our game environment with the following parameters: one airport destination, no obstacles, 25 planes, and a varying number of episodes. Each episode is a run of the program where the agent accumulates points until two planes collided.

We compared the performance of our agent against two benchmarks - a random policy agent and a straight line following agent. Figures 3-6 show the comparison between the performance of the agents.

B. Performance

1) Random Policy Agent: We used a random policy following agent as one of our bench-marking agents. This agent takes one of the possible actions at random to avoid collisions when near other planes. As we can see in Fig. 3 , this agent shows a peak score in the beginning. Since, the airplane spawn events are random; this introduces some chance events with very sparse plane density. This might be an explanation for an alienated spike seen in the beginning. We count this as the ”burn-in” time of our environment.

We can see that the scores average to around 30 for the random policy event as random policy should produce similar scores when averaged over a bunch of episodes. Moreover, the reason for these low scores is that the random policy agent doesn’t learn to improve its results over time.

2) Straight Line Following Greedy Agent: We used a straight line following agent as another of our bench-marking greedy agents. This agent takes the straight line trajectory to their destination regardless of the impending collisions. As we can see in Fig. 4, this agent shows some peak scores during its run. This can again be attributed to the burn-in time of our

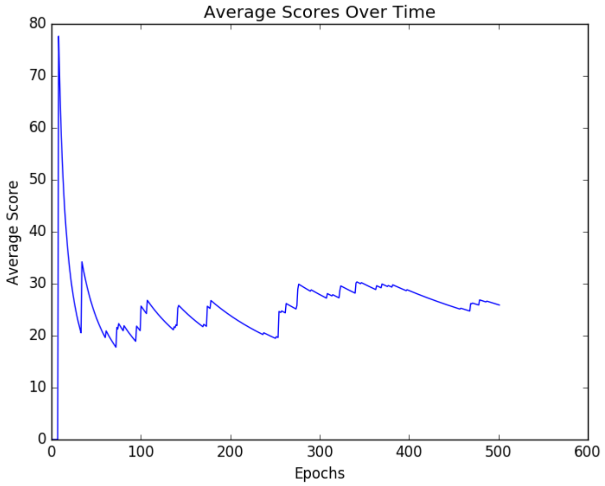


Fig. 3. Average scores for baseline random agent over 500 episodes.

environment. These spawn events happen at random and are sparsely distributed across the curve.

We can see that the scores average to around 100 for this agent as this agent is focused on reaching the destination without caring about the collision. Due to this aggressive maneuver; many times one or two planes in the episodes reach their destination; raising the score above the random policy agent's score. Thus, this greedy agent aims to maximize reward but fails to do better than our agent because of the failure to learn from the collisions.

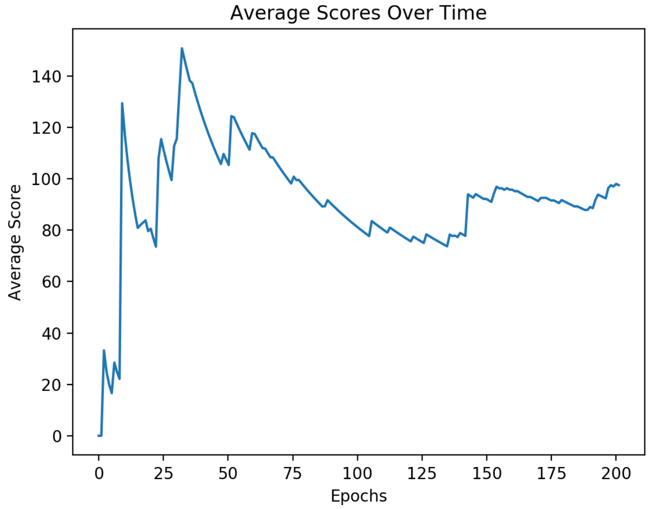


Fig. 4. Average scores for baseline greedy agent over 500 episodes.

3) SARSA Agent: We trained our SARSA agent with 25 planes to follow a single airport destination. Table III below illustrates the different parameters used while training the SARSA agent.

TABLE III
SARSA AGENT PARAMETERS

Fig #	# of planes	α	γ	exploration probability
5	25	0.5	0.9	0.5 (reduces over time)
6	25	0.5	0.9	0.1

First, we trained the agent for 125 episodes with 25 planes, a learning rate of 0.5 and a discount factor of 0.9. In this case, we started with an exploration probability of 0.5; which we reduced by 10% every 10 episodes. The reason for doing this is that we wanted to start with more exploration and gradually reduce the exploration to start exploiting the knowledge gained from our Q-Tables.

As you can see in Fig. 5, our SARSA agent has an average score of around 300, which is 10 times better than the average score for the random agent and about 3 times better than the greedy agent.

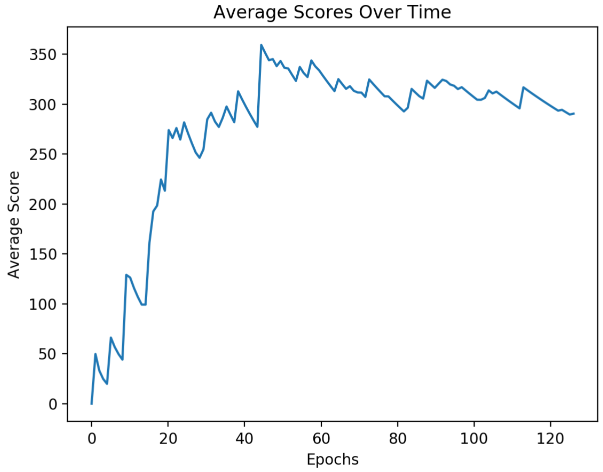


Fig. 5. Average score for SARSA agent over 125 episodes.

Following the previous training instance; we trained the agent for 500 episodes with 25 planes, a learning rate of 0.5 and a discount factor of 0.9. In this case, we used an exploration probability of 0.1; since we wanted to aggressively exploit our learned policy.

As you can see in Fig. 6, our SARSA agent converges to an average score of about 300 with a steadily increasing trend for scores. This shows again that our SARSA agent is 10 times better than the average score for the random agent and about 3 times better than the greedy agent.

TABLE IV
AVERAGE SCORES OF VARIOUS AGENTS

Agent	Average Score
baseline - random agent	30
baseline - greedy agent	100
sarsa agent	300

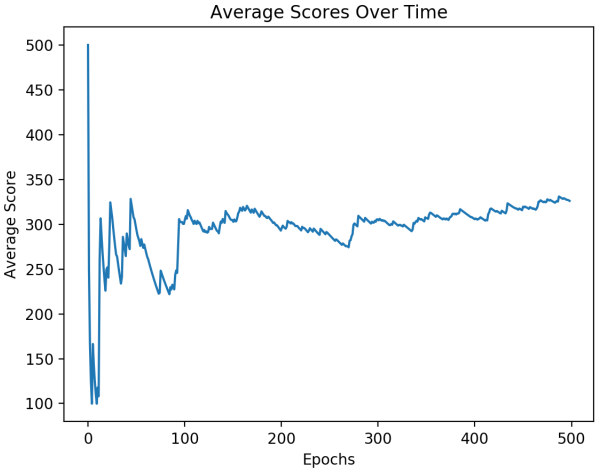


Fig. 6. Average score for SARSA agent over 500 episodes.

VIII. Conclusion

In this paper, we presented a reinforcement learning approach to designing an aircraft collision avoidance system similar to what is used in air traffic control. We used a PyGame environment to simulate a simplified version of this scenario and modeled our agent as an MDP running a SARSA algorithm. Our SARSA agent was able to achieve reasonable success in collision avoidance compared to our baseline agents. Our RL agent had consistently better scores once it had learned from its initial episodes.

There still remains a lot of room for improvement and development for our agent. We can add actions to our agents such as an option to reduce or increase the speed along with an option to change directions. Also, we did not have enough time to run with various configurations, which we had planned with our simulator. Future work might involve training our RL agent with different obstacles and multiple airport destinations with often crossing paths.

There have been a lot of recent advances in Deep Reinforcement Learning as discussed in the lectures [5]. It would be interesting to observe how our aircraft collision avoidance model performs under deep neural network methods for reinforcement learning. Another extension for our project could be to model our agent using a POMDP approach instead of an MDP by introducing state uncertainty along with noisy measurements.

Our current implementation involves simulation based learning, which we may modify in the future to learn without running the simulation; thus, making the training process faster. Currently, we are limited by the frame rendering rate for running episodes. We could also try other RL algorithms instead of SARSA, or even implement SARSA-Lambda to increase reward propagation. An interesting learning case could be to train our agent alongside the greedy agent to improve our agent's performance against greedy and smart agents at the same time.

IX. Contributions

Initial project idea, design, and scoping was pioneered by Aamir. John and Ashar implemented the SARSA reinforcement learning algorithm in Python and wrote a majority of the paper. Ron and Aamir modified the PyGame environment, with Ron handling the majority of the required programming. All members trained either a baseline model or SARSA agent.

References

[1] Julian, Kyle & Lopez, Jessica & Brush, Jeffrey & Owen, Michael & Kochenderfer, Mykel. (2016). Policy compression for aircraft collision avoidance systems. 1-10. 10.1109/DASC.2016.7778091.

[2] M. J. Kochenderfer and N. Monath, "Compression of optimal value functions for Markov decision processes," in Data Compression Conference, 2013.

[3] "News," pygame.org. [Online]. Available: http://www.pygame.org/. [Accessed: 04-Dec-2019].

[4] https://github.com/scotty3785/python-air-traffic-control

[5] M. J. Kochenderfer, "Model Uncertainty" in Decision Making under Uncertainty: Theory and Application, MIT Press, 2015

[6] https://github.com/aamirrasheed/air-traffic-control-AI