Planning for Air Traffic Control*

Ben Potter[1]

***Abstract*— Air traffic control services face increasing demand while recruitment and training of new personnel remains slow. This work explores autonomous planning for air traffic control by implementing and comparing graph-based and sampling-based motion planning algorithms. I model aircraft using the Dubins airplane model and formalize the planning problem with a learned penalty function that encodes human controller behavior from historical flight trajectories. I implement two variants each of Anytime Repairing A* and Rapidly-exploring Random Tree using path length and motion cost metrics. The algorithms are evaluated on 1,978 trials derived from real flight trajectories at Seattle-Tacoma Airport. Results show that Anytime Repairing A* successfully solves 96.1% of trials within 180 seconds, while Rapidly-exploring Random Tree struggles to reach small goal regions reliably, solving only 0.1% of motion cost trials. These findings suggest that heuristic-guided search is essential for planning problems with tight kinodynamic constraints and small goal regions. The learned penalty function successfully captures common flight routes and provides a principled approach for incorporating human expertise into automated air traffic control systems.**

I. INTRODUCTION

Air traffic control services are overwhelmed by current demand, which is only expected to increase due to the emerging feasibility of aerial robotics. Additionally, recruitment and training of new air traffic control personnel is slow and strict regulations limit the size of the eligible workforce. Safe and reliable air traffic control requires experienced personnel who can follow the complex set of regulations, including requirements on spacing between airplanes, weather restrictions, and unique departure and arrival protocols at different airports. These regulations often require air traffic control personnel to formulate complex strategies that opaquely optimize across all of these factors.

I propose constructing an autonomous air traffic control operator that could drastically reduce the load on human personnel. The primary challenge in developing such a system is formalizing the implicit knowledge that human controllers use when planning trajectories. Human controllers do not simply optimize for the shortest path; they account for numerous safety constraints, weather patterns, traffic flow, and airport-specific procedures that are difficult to encode explicitly. This makes the problem of finding the best path fundamentally challenging, since the optimal solution must balance path length against these implicit constraints.

To address this challenge, I use a learned penalty function that encodes human behavior from historical flight trajectories. This penalty function is incorporated into the cost metric used by motion planning algorithms. I implement and compare two planning approaches: Anytime Repairing A*, a graph-based planner, and Rapidly-exploring Random Tree, a sampling-based planner. The Anytime Repairing A* approach was originally proposed by Tolstaya et al., and I adapt it for use in this work [**?**].

The contributions of this work are threefold. First, I present an offline method for learning the penalty function from human trajectories that differs from prior inverse reinforcement learning approaches. Second, I develop and compare implementations of both graph-based and sampling-based planners on the air traffic control problem using real flight data. Third, I demonstrate that sampling-based methods face significant challenges in this domain due to the combination of non-holonomic constraints and small goal regions, which suggests that heuristic-guided search may be essential for this class of problems.

The remainder of this paper is organized as follows. Section **??** provides background on the Dubins airplane model and formalizes the planning problem. Sections **??** and **??** describe the implementation details of Anytime Repairing A* and Rapidly-exploring Random Tree, respectively. Section **??** explains the learned penalty factor, and Section **??** describes the dataset processing pipeline. Section **??** outlines the experimental design, and Section **??** presents the results. Finally, Section **??** concludes with a discussion of future work.

II. BACKGROUND

I model the state of an airplane using the Dubins airplane model, a four-dimensional system with a configuration space, $S = \mathbb{R}^3 \times [-\pi, \pi)$, with $s = (x, y, z, \theta)$. In this model, x, y, and z describe the coordinates of the airplane in three-dimensional euclidean space, and θ is the bearing of the airplane relative to the $+x$ axis. The spatial dimensions of the state space are described using East, North, Up conventions. The convention aligns the x, y, and z axes with the east, north, and up directions, respectively.

The system is controlled through the first derivatives of altitude and bearing, $\dot{z}$ and $\dot{\theta}$, with control inputs denoted as u_z and u_θ, respectively. The model assumes the aircraft has a fixed speed in the xy-plane which is denoted v. The motion of the airplane is described by Eqn. **??**.

$$\dot{s} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} v\cos\theta \\ v\sin\theta \\ u_z \\ u_\theta \end{bmatrix} \tag{1}$$

*This work was completed for ELEC 844 taught by Jonathan Gammell

[1] Department of Electrical and Computer Engineering, Queen's University, Kingston, Canada ben.potter@queensu.ca

The planning problem is the problem of finding the lowest cost path between a starting state s_0 and a goal state s_g. These states are given to each airplane by a human air traffic controller when the airplane approaches an airport. The goal state corresponds to the location and bearing of a runway where the airplane should land. I assume that this information is available prior to planning.

One of the challenges with formalizing the planning problem is quantifying the true cost of a trajectory. I start by providing the formalization for the naive path length cost metric. This metric is naive because it ignores many of the safety constraints that human experts must consider. In other words, the shortest path trajectory is often not the best. Then, I describe a formalization based on a learned penalty function which aims to encode these extraneous constraints in a cost function that can be minimized. The formalization of the planning problem for air traffic control I use is first presented by Tolstaya et al. [**?**].

The path length cost metric is based on the Dubins distance. The Dubins distance is the length of the Dubins curve which is the shortest path between two states. The Dubins distance is well-defined for the more common wheeled robot configuration space that does not have a z component, but I extend the definition to the Dubins airplane model using methods from the literature.

The Dubins curve depends on the kinodynamic constraints imposed on the Dubins airplane model. In other words, the kinodynamic constraints place bounds on allowable values of u_z and u_θ. In this work, I use $|u_z| \leq 6$ meters per second and $|u_\theta| < 0.025$ radians per second. Following Eqn. **??**, $u(t)$ is in U iff these constraints are met.

$$\begin{aligned} &\arg\min_{s(t)} \int_{t_0}^{T} \|\dot{s}(t)\|\, dt \\ \text{s.t. } &\dot{s}(t) = \big(\cos\theta, \sin\theta, u_z(t), u_\theta(t)\big) \\ &s(t_0) = s_0, s(T) = s_g \\ &s(t) \in S, u(t) \in U \end{aligned} \tag{2}$$

I incorporate a penalty factor $J(s)$ into the cost function by adding an additional term which is the product of the penalty factor and the path length through which that penalty factor is incurred. The procedure for learning the penalty is discussed in Section **??**. For example, if the penalty factor at a state s' is 1, then the cost to reach s' from a state s effectively double the path length between s and s'. This variant of cost function is called the motion cost. The planning problem is re-formalized in Eqn. **??** using the motion cost.

$$\begin{aligned} &\arg\min_{s(t)} \int_{t_0}^{T} (1 + J(s)) \|\dot{s}(t)\|\, dt \\ \text{s.t. } &\dot{s}(t) = \big(\cos\theta, \sin\theta, u_z(t), u_\theta(t)\big) \\ &s(t_0) = s_0, s(T) = s_g \\ &s(t) \in S, u(t) \in U \end{aligned} \tag{3}$$

Rather than requiring that $s(T) = s_g$ directly in Eqn. **??** and Eqn. **??**, I introduce a region of states which count as the goal. I define the goal region as any state within 0.5 kilometers for the xy axes, 0.25 kilometers for the z-axis, and 0.125 radians for the θ-axis bounds of s_g.

From these two definitions of the planning problem, I derive four different planners which I study in this work. I implement two variants of Anytime Repairing A* which use the path length and motion cost cost functions and call them Path Length Anytime Repairing A* and Motion Cost Anytime Repairing A*, respectively. Similarly, I implement two variants of Rapidly-exploring Random Tree called Path Length Rapidly-exploring Random Tree and Motion Cost Rapidly-exploring Random Tree.

III. ANYTIME REPAIRING A*

The Anytime Repairing A* planner operates over a graph representation of the configuration space. It works by iteratively expanding the current minimum cost states which have the highest likelihood of being on the optimal path to the goal. The rules for moving between states are defined by a set of fixed-time motion primitives that are derived from the kinodynamic constraints of the airplane. In the following section, I explain my implementation of Anytime Repairing A* including the configuration space discretization, the fixed-time motion primitives, and the cost function. My implementation follows the example presented in [**?**].

Anytime Repairing A* is a graph-based planner which requires a discretization of the configuration space. I discretize each dimension of the configuration space according to a minimum resolution $\rho \in \mathbb{R}^4$. Given a continuous state s, its discretized state $\bar{s}$ is given by Eqn. **??**. I choose the minimum resolution $\rho_x = \rho_y = 0.125$ kilometers, $\rho_z = 0.05$ kilometers, and $\rho_\theta = 0.05$ radians. The discretized configuration space is still too large to store in memory, so I construct the graph lazily as the planner explores. This will impact runtime as the planner needs to perform memory allocations on-the-fly.

$$\bar{s} = \left[\left\lfloor \frac{s_x}{\rho_x} \right\rfloor, \left\lfloor \frac{s_y}{\rho_x} \right\rfloor, \left\lfloor \frac{s_z}{\rho_x} \right\rfloor, \left\lfloor \frac{s_\theta}{\rho_\theta} \right\rfloor \right] \tag{4}$$

The fixed-time motion primitives define the allowable edges between states in the graph. I choose to fix the time interval of every edge to $\Delta t = 30$ seconds. From there, I construct a set of allowed control inputs for the altitude and bearing. The altitude change is chosen from the set $u_z \in \{-\Delta z, 0, \Delta z\}$ and the bearing from the set $u_\theta \in \{-\Delta\theta, 0, \Delta\theta\}$. I select $\Delta z = 6$ meters per second and $\Delta\theta = 0.025$ radians per second. These values are representative of the typical control inputs available to modern commercial airplanes.

The cost of a state, denoted $f(s)$, is defined as the sum of two other distinct costs: the cost-to-come, denoted $g(s)$ and the cost-to-go, denoted $h(s)$. The cost-to-come to a state s represents the cost of the best path (so far) from the starting state to s. The cost-to-come is computed using the path length and motion cost cost functions. The cost-to-go for a state s is the expected remaining cost of the optimal path from s to the goal state.

Algorithm 1 Dubins Airplane Heuristic

Input: Start s_0, goal s_g, forward speed v, max rate of climb Δz, turning rate $\Delta\theta$

Output: Minimum path length from start to goal, d_{min}

1: Compute Dubins car distance d_{xy} using the LSL, RSR, LSR, RSL, RLR, LRL paths with curvature $\kappa = \Delta\theta / v$

2: Compute the minimum time from start to goal in the xy plane

$$t_{min} = d_{xy}/v$$

3: Compute the minimum time for ascent/descent

$$t_z = |z_g - z_0|/\Delta z$$

4: **while** $t_z > t_{min}$ **do**

5: Add a helical ascent/descent to the trajectory

$$t_{min} = t_{min} + \frac{2\pi}{\Delta\theta}$$

6: **end while**

7: Compute the minimum path length from start to goal:

$$d_{min} = \sqrt{(v \cdot t_{min})^2 + (z_g - z_0)^2}$$

The procedure for computing the cost-to-go is defined in Algorithm **??**. The algorithm considers the xy and z motions of the airplane separately. It starts by computing the Dubins curve in the xy-plane. Then, it uses the constant velocity of the airplane in the xy-plane to find the minimum time to traverse the Dubins curve. Algorithm **??** then attempts to complete any z motion within the minimum time. If the minimum time is too short, then helical sections are added to the trajectory which allow the airplane to descend to the same xy position and try to complete the z motion again. This approach works for most trials, but it is incorrect in cases where a helical descent is not feasible.

When summing the two distinct costs, the cost-to-go can be inflated by a factor ϵ to encourage the algorithm to move towards the goal as quickly as possible. This method yields feasible solutions quickly, but such solutions may not be optimal. In many situations, it is desirable for an algorithm to yield an initial solution quickly. Then, given more time, progressively improve the solution. The Anytime Repairing A* algorithm does this by iteratively deflating the factor ϵ until the solution is optimal. In this paper, I initialize each trial with $\epsilon = 3.0$.

IV. RAPIDLY-EXPLORING RANDOM TREE

My implementation of Rapidly-exploring Random Tree has three major procedures: sampling, steering, and rewiring. The sampling procedure generates new states which explore the state space looking for the goal. The steering procedure decides how the algorithm should incorporate new states into its current search tree. The rewiring procedure ensures that the current search tree will approach optimality as the steering procedure incorporates additional states. In the following section, I explore each of these procedures and the different implementations that I consider.

A. Sampling

I start by considering sampling based on the uniform distribution, but I find that this approach spends too much time exploring unproductive regions. The search space is large which means the density of samples along the path to the goal is low. As such, I consider non-uniform sampling approaches using the penalty and the Euclidean distance to the goal. The aim of these approaches is to increase the density of samples in useful regions of the search space.

The non-uniform sampling procedure will sample the goal with $p_g = 0.1$, or it will sample uniformly with $p_u = 0.1$. Otherwise, it will sample according to the non-uniform distribution. I choose $p_u > 0$ to ensure that the algorithm still explores the full search space however infrequently. I use the Metropolis-Hastings algorithm to sample from a non-uniform distribution. While testing, I considered a rejection sampling approach, but I found that this inflated the runtime to generate a sample significantly. This is likely because the high probability regions of the distribution are much smaller than the size of the search space which leads to many rejected samples.

The Metropolis-Hastings algorithm is used to sample a state s from any probability distribution $P(s)$ using a function $f(s)$ that is proportional to P. It starts by choosing an arbitrary state s_t and a *proposal function* $g(s'|s_t)$. Then, for each iteration t, the algorithm proposes a candidate sample s' by picking from the distribution $g(s'|s_t)$. The candidate sample has an *acceptance ratio* $\alpha = f(s')/f(s_t)$. The sample is accepted with the probability α. If the sample is accepted, the algorithm sets $s_{t+1} = s'$. Otherwise, it sets $s_{t+1} = s_t$. The algorithm iterates over candidate samples until one is accepted at which point it proceeds with the steering procedure.

The proposal function takes a state s_t and returns a state s' according to an offset sampled from a standard Gaussian.

$$g(s_t) = s' = s_t + \mathcal{N}(0, 1)$$

For the distribution function f, I use an exponential weighted by a parameter β. The exponential is evaluated according to a function $h(s)$ which provides the underlying distribution. The β parameter controls the importance of the underlying distribution against the proposal function when generating samples. A larger value of β will guide the generated samples towards the lower values of the distribution more strongly. I negate the β parameter because my underlying distributions $h(s)$ are inverted. In other words, I want to sample smaller values of $h(s)$ more frequently. I use $\beta = 8$ in my implementation. I test two different versions of $h(s)$ based on the penalty factor and the Euclidean distance to the goal.

$$f(s) = \exp(-\beta h(s))$$

1) Penalty distribution: In the penalty distribution, I set $h(s) = J(s)$. The idea with this distribution is to exploit the fact that the motion cost depends on the penalty factor. In order to minimize the motion cost, and thus the penalty factor I incur, it makes sense to increase the sampling density in regions of the configuration space with a low penalty factor.

2) Distance-to-goal distribution: In the distance-to-goal distribution, I set $h(s)$ to the Euclidean distance from s to the goal s_g. I test this distribution because I want the sampling procedure to minimize the number of samples generated close to the boundaries of the configuration space. The goal state s_g is always near the center of the configuration space, so sampling near the boundaries is often unproductive.

B. Steering

The steering procedure is defined using the Dubins curve and parameterized by the maximum edge length, denoted η. I refer to η as the time it takes to traverse an edge rather than the length of the edge. Given a sampled state s', the procedure begins by finding the nearest state s'' to s' that has already been expanded. The proximity of two states is measured using the Dubins distance. It is possible that s' is not directly reachable from s''. The next step is to generate a new state s from s'' towards s'. I start by computing the Dubins curve from s'' to s' ignoring any difference in the z components. Then, I find the furthest state s along the Dubins curve which is reachable in at most η seconds. This could be s' (i.e., the endpoint of the Dubins curve), but is typically some intermediate state which is on the shortest xy path from s''. At this point, the new state s has its x, y, and θ components defined, but it is missing any motion along the z-axis. Given the time t to traverse the Dubins curve from s'' to s in the xy-plane, I calculate the z component of the new state to be the closest to s' the airplane can get in t seconds without exceeding the maximum altitude speed. This steering procedure returns the endpoint of the segment of the shortest trajectory between s'' and s' which takes at most η seconds and does not violate the kinodynamic constrains of the Dubins airplane model. The new state s is wired into the tree as part of the rewiring procedure.

C. Rewiring

The rewiring procedure ensures that the current search tree will approach optimality as the steering procedure incorporates additional states. It does this by optimizing first order edges for each new state within the state's neighborhood. Optimizing the first order edges starts by choosing the lowest cost parent and then adopting any neighboring states that would benefit (i.e., their cost would decrease). A neighborhood is computed using the k-nearest neighbors algorithm where k is defined by Eqn. **??**. The dimensionality of the configuration space n is 4, and the cardinality of the search tree $|V|$ grows as the algorithm adds states.

$$k > e(1 + 1/n)\log(|V|) \tag{5}$$

The method of computing the k-nearest neighbors has a significant impact on the performance of Rapidly-exploring Random Tree. I test three different distance metrics with k-nearest neighbors that are based on the euclidean distance and the Dubins distance. The *Euclidean* metric ranks states by their euclidean distance which ignores the kinodynamic constraints of the airplane. This approach dilutes the effect of local optimization by underestimating the true distance between neighbors. However, the euclidean distance is fast to compute which means Rapidly-exploring Random Tree can spend more time searching. Alternatively, the *Dubins* metric computes the length of the shortest path that obeys the kinodynamic constraints of the airplane between neighbors. This approach is slower to compute than the euclidean distance. The Dubins distance, unlike the euclidean distance, is not commutative (i.e., the Dubins distance from a to b is not always the same as the Dubins distance from b to a). This requires computing the k-nearest neighbors for the candidate parents and candidate children of each new state separately. Finally, I consider a combination of these metrics called *Dubins + Filter* whereby the fast to compute euclidean distance is used to filter the search tree down to the closest $2k$ states, before the Dubins distance selects the best k states. This metric aims to use the speed of the Euclidean metric while maintaining the tight bound provided by the Dubins metric.

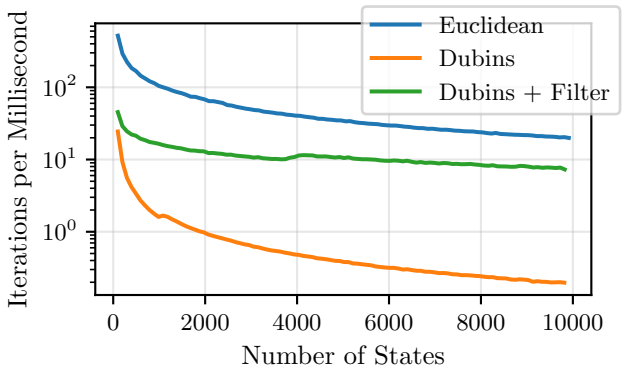

Fig. 1. A plot comparing the performance of three k-nearest neighbors metrics used in Rapidly-exploring Random Tree. The x-axis shows the number of states that Rapidly-exploring Random Tree has expanded. The y-axis shows the number of iterations that Rapidly-exploring Random Tree completes per millisecond in log scale.

I compare the performance of Rapidly-exploring Random Tree using different k-nearest neighbor metrics in Fig. **??**. As a measure of performance, I consider the number of iterations that Rapidly-exploring Random Tree completes per millisecond which is shown on the y-axis in log scale. I show how this metric changes as the number of states in the search tree increases which is shown on the x-axis. Each execution of Rapidly-exploring Random Tree is given the same start and goal states and performs 10,000 iterations. The Dubins + Filter metric seems to provide a runtime comparable to the Euclidean metric, even as the number of states grows large, while maintaining the tight bound of the Dubins metric. In the remaining results of this paper, I use the Dubins + Filter metric to find the k-nearest neighbors in

Rapidly-exploring Random Tree.

I use a k-dimensional tree to increase the speed of the filter in the Dubins + Filter metric. The working principle of the k-dimensional tree is to store expanded states according to their location in state space, which makes spatial lookups fast. This method is often used in implementations of k-nearest neighbors since it does not require searching over all expanded states.

When Rapidly-exploring Random Tree optimizes neighbors it must ensure that any new edges are within the maximum edge length η and that they obey the kinodynamic constraints of the airplane. I do this by computing the Dubins curve for a starting state s and a target state s' in the xy-plane. Given the length of the Dubins curve and the constant velocity of the airplane, I find the time to traverse the length. If the time to traverse the length is larger than η, the path is not feasible and is discarded. This is because the Dubins curve is the shortest path between s and s' so all other paths must take at least as long to traverse. Then, I handle any changes in altitude between s and s'. With the time to traverse the Dubins curve in the xy-plane, I find the maximum possible altitude change using the upper bound on u_z. If the altitude change between s and s' does not exceed the maximum possible alitude change, then the edge is accepted as feasible.

V. PENALTY FACTOR

The penalty factor is learned from human guided trajectories. Broadly speaking, the penalty factor for a state s is on the range $[0, 1]$ and is inversely proportional to the number of times a human trajectory from the dataset includes s. The penalty factor can be thought of as the cost of diverging from the beaten path. This penalty factor is intended to encode the behavior of human guided trajectories in solutions returned by automated planners.

The penalty is computed on a discretized state space. Following the example from [**?**], I use a coarser discretization for the penalty than the one used by Anytime Repairing A*. The x and y axes are $\rho_x = \rho_y = 0.5$ kilometers with the z axis as $\rho_z = 0.25$ kilometers. The θ axis is $\rho_\theta = 0.25$ radians. To get a discretized state $\bar{s}$ from a continuous state s, I use Eqn. **??**.

I learn the penalty factor by recording a count of the number of times that a human guided trajectory passes through each discretized state. Then, I normalize the counts onto the range $[0, 1]$. After normalization, I apply a gamma correction of $\gamma = 0.25$ which increases the importance of discretized states with a small count. I use gamma correction because there are many discretized states that have small counts only because of the size of the discretization. Conversely, the discretized states close to the runway, which are traversed by all trajectories, have very large counts. This makes the simple normalization insufficient for capturing a useful penalty for the full state space. Finally, I invert and scale the count to get the final penalty.

The penalty factor for a discretized state $\bar{s}$ is shown in Eqn. **??**. The function $norm(\bar{s})$ describes the normalized count for the discretized state $\bar{s}$. If there are no human guided trajectories which pass through $\bar{s}$ (i.e., $norm(\bar{s}) = 0$), then $J(\bar{s})$ is 1. For $\bar{s}$ with $norm(\bar{s}) > 0$, the penalty factor is on the range $[0, \frac{1}{2}]$.

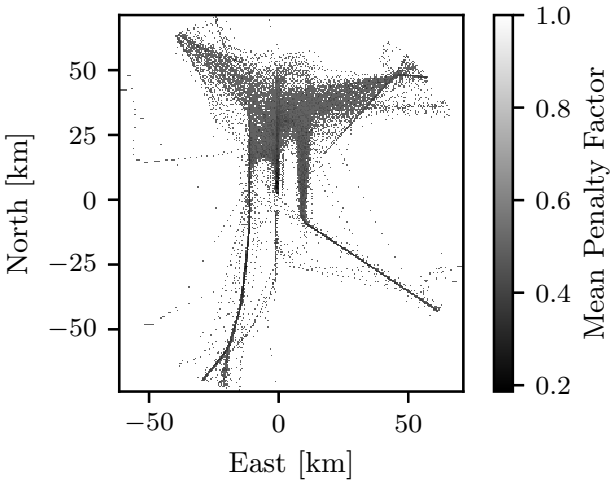

Fig. 2. A heatmap showing the mean penalty factor across the x and y axes of the configuration space. The x-axis represents the x position in kilometers of each state with respect to Seattle-Tacoma Airport using the East, North, Up convention. The y-axis represents the y position in kilometers of each state with respect to Seattle-Tacoma Airport using the East, North, Up convention.

$$J(\bar{s}) = \begin{cases} 1 & \text{if } norm(\bar{s}) = 0 \\ \frac{1}{2}(1 - norm(\bar{s})^\gamma) & \text{otherwise} \end{cases} \quad (6)$$

I plot the penalty factor for each discretized state from a view looking down the z axis in Fig. **??**. I project the state space into 2 dimensions by taking the mean of the penalty factor $J(\bar{s})$ over the z and θ axes for each x and y coordinate. The x and y axes are shown using the East, North, Up convention. The mean penalty factor is expressed as a grayscale color with lighter coordinates indicating a higher value. The darker lines in Fig. **??** represent common routes taken by human guided flights.

The approach I take to learn the penalty factor is different from the approach presented in [**?**]. Tolstaya et al. use maximum entropy inverse reinforcement learning which updates the penalty according to gradient descent each time a new solution is found by Anytime Repairing A*. Further, their penalty factor includes an inter-airplane safety cost that extends their solution to handle multiple agents. This approach seems out-of-scope for the research presented in this paper, so my alternative formulation, which is completely offline, is used instead. I made heuristic comparisons with a figure from [**?**] that is similar to Fig. **??** that suggest our penalty factors are comparable despite the differences in our approaches.

VI. DATASET PROCESSING

I collect a dataset of human guided trajectories for benchmarking the solutions returned by the planners and constructing the learned penalty factor. The dataset contains real

flight trajectories recorded during their landings at Seattle-Tacoma Airport between January 11 and 14, 2016. The raw data for each flight is downloaded from FlightAware. The raw data consists of World Geodetic System positions and heading measurements. I process the raw data into a sequence of states which use the Dubins airplane model. For each trajectory, I compute the path length and the motion cost as if they were solutions returned by a planner.

I start by pruning the start of each flight trajectory until it is contained in the search space. I define the search space using the great-circle distance with reference to the World Geodetic System position of Seattle-Tacoma Airport. All positions outside a 75 kilometer great-circle centered at Seattle-Tacoma Airport are excluded from the flight trajectory. After pruning, I map the raw data from each trajectory into a sequence of states. The World Geodetic System positions are converted to spatial coordinates using the East, North, Up convention. The origin of the East, North, Up reference frame is the World Geodetic System center of Seattle-Tacoma Airport. The headings in the raw data are consistent with the Dubins airplane model. A state is constructed from each East, North, Up position and heading.

The dataset is designed to follow the example provided in the work by Tolstaya et al. with a few differences [**?**]. My dataset contains flights from January 14, 2016 which are not included in the previous work. The previous work does not discuss the size of their search space, so I estimate it to be around 75 kilometers using their figures.

VII. EXPERIMENTAL DESIGN

I test four planner variants in my experiments and I explain their configurations here. For the two Anytime Repairing A* variants, the only difference in their configuration is their definition of cost-to-come. Path Length Anytime Repairing A* uses a cost-to-come based on the path length metric, but Motion Cost Anytime Repairing A* uses a cost-to-come based on the motion cost metric. The two Rapidly-exploring Random Tree variants follow the same cost-to-come definitions as their Anytime Repairing A* counterparts, but they have more configuration differences. The Path Length Rapidly-exploring Random Tree planner uses a maximum edge length of 180 seconds with a uniform sampling procedure. The Motion Cost Rapidly-exploring Random Tree planner uses a maximum edge length of 30 seconds with the distance-to-goal sampling procedure.

Each planner variant is tested on 1,978 trials derived from human guided trajectories. A trial consists of a starting state and a goal state which correspond to the first and last states in each human guided trajectory. For each trial, I measure the number of iterations, the number of states, and the cost of the best solution found. For path length and motion cost variants the cost is the path length and motion cost, respectively. I record each metric at 30, 60, and 180 seconds of runtime. I measure Anytime Repairing A* metrics each time the solution is improved, which may not align with recording period, so recorded metrics correspond to the best solution found at the time of recording. The measuring approach for Rapidly-exploring Random Tree is more flexible as I can control the exact number of iterations. I record metrics after every 5,000 iterations. I execute trials on the High-Performance Compute cluster provided by Centre for Advanced Computing. The trials are run on a machine with 64 cores and 976 gigabytes of memory. I allow trials to execute in parallel using the GNU Parallel tool [**?**].

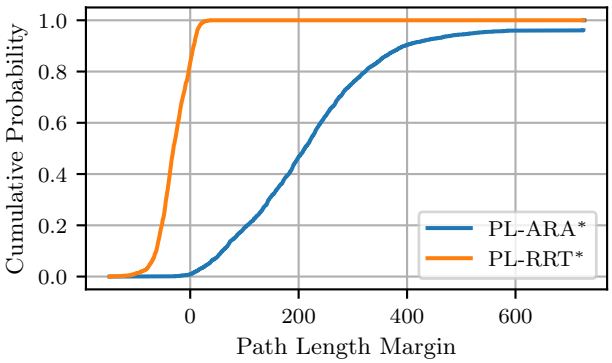

Fig. 3. A plot of the empirical cummulative density function comparing the cost margin by path length of the Path Length Anytime Repairing A* and Path Length Rapidly-exploring Random Tree planners. The x-axis represents the cost margin by path length. The y-axis represents the probability that a solution has a certain cost margin based on the experimental results for all trials.

With the results collected from each trial, I compute five statistical metrics for each planner variant. I find the median number of iterations, the median number of expanded states, the median cost, and the median cost margin for all planners. For path length and motion cost variants, the cost and cost margin use the path length and motion cost, respectively. The cost margin is the difference between the cost of the planner solution and the cost of the human trajectory for the same trial. I also find the percent of trials, for each planner, which return a solution. All the statistical metrics are computed after each runtime period.

VIII. RESULTS

My results indicate that Anytime Repairing A* is better at finding solutions to the planning problem than Rapidly-exploring Random Tree. Anytime Repairing A* performs better in both the path length and motion cost variants of the planning problem. Rapidly-exploring Random Tree fails to return a solution within the runtime limit in most cases because it struggles with expanding states near the goal region. In the following section, I present my experimental results and I explain how I interpret my findings from them.

Starting with the path length variant of the planning problem, I consider the cost margin by path length for Path Length Anytime Repairing A* and Path Length Rapidly-exploring Random Tree in Fig. **??**. Fig. **??** shows an empirical cummulative distribution function for the cost margin by path length. The x-axis shows values of the cost margin by path length and the y-axis shows the cumulative probability of observing that cost margin. The cost margins from

TABLE I

EXPERIMENTAL RESULTS FROM THE ATC PLANNING PROBLEM WITH ANYTIME REPAIRING A* AND RAPIDLY-EXPLORING RANDOM TREE VARIANTS.

Planner	**Runtime [s]**	**Median Iterations**	**Median States**	**Median Path Length**	**Median Cost Margin**	**% Solved**
Path Length Anytime Repairing A*	30	213,482.0	1,038,534.0	420.1	314.4	65.4
	60	531,878.0	2,589,844.0	363.1	254.5	82.1
	180	1,248,564.0	5,533,595.0	320.7	211.4	96.1
Path Length Rapidly-exploring Random Tree	30	53,000.0	47,694.0	81.9	−31.0	100.0
	60	79,000.0	71,092.0	81.9	−31.0	100.0
	180	136,000.0	122,535.0	81.9	−31.0	100.0
Human				111.3		
Planner	**Runtime [s]**	**Median Iterations**	**Median States**	**Median Motion Cost**	**Median Cost Margin**	**% Solved**
Motion Cost Anytime Repairing A*	30	0.0	0.0	inf	inf	25.8
	60	0.0	0.0	inf	inf	44.3
	180	2,161,474.0	9,381,285.0	735.8	629.1	79.8
Motion Cost Rapidly-exploring Random Tree	30	365,000.0	20,392.5	inf	inf	0.0
	60	635,000.0	43,617.0	inf	inf	0.1
	180	1,500,000.0	150,281.0	inf	inf	0.1
Human				2,322.0		

solutions found by Path Length Anytime Repairing A* are positive which indicates that its solutions are consistently longer than the human examples. This is expected in part because the planner does not fully deflate ϵ within the runtime limit. Conversely, Path Length Rapidly-exploring Random Tree reports many negative cost margins. The 0.8 quantile of Path Length Rapidly-exploring Random Tree is 0 which suggests that it frequently finds shorter trajectories than human examples. These results are caused by the maximum edge length configured for Path Length Rapidly-exploring Random Tree.

If the maximum edge length of Rapidly-exploring Random Tree puts the goal region within the range of a single edge from the start state, then Rapidly-exploring Random Tree collapses to returning the Dubins curve. In Path Length Rapidly-exploring Random Tree trials, the maximum edge length is equivalent to 180 seconds of travel time which puts the goal region in range for most trials. The effect of this parameter is shown in Tab. **??** where the median path length does not improve with additional runtime and it has a negative cost margin. The median path length does not improve with additional runtime because it finds the absolute shortest possible path on the first iteration that the goal state is sampled. The negative cost margin indicates that the solution returned by Path Length Rapidly-exploring Random Tree is shorter than the human guided trajectory. The results show a negative cost margin because the Path Length Rapidly-exploring Random Tree path is always the absolute shortest possible path that obeys the kinodynamic constraints of the airplane. The shortest possible path is often not feasible in practical scenarios where additional safety and higher order kinodynamic constraints are present. The Dubins airplane model does not encode these additional constraints so the minimal path length solution is not always the best path. Therefore, it is reasonable to consider reducing the maximum edge length to prevent this case and force Path

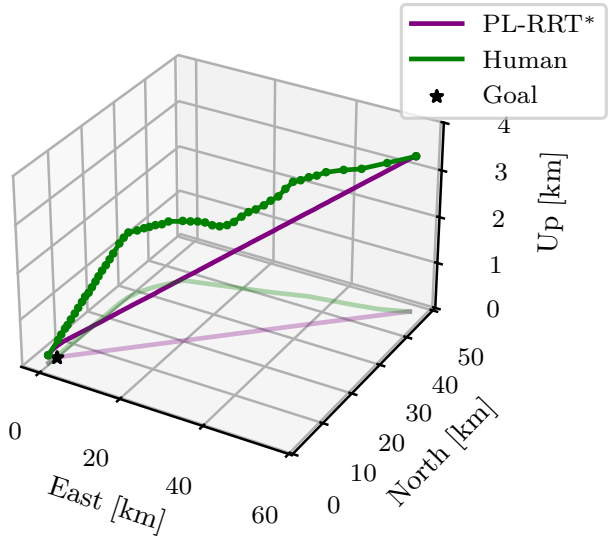

Fig. 4. A trajectory plot comparing a solution found by Path Length Rapidly-exploring Random Tree and the human guided reference. The goal region is denoted with a star. The plot axes are expressed using the East, North, Up convention with the origin at Seattle-Tacoma Airport.

Length Rapidly-exploring Random Tree to generate paths with multiple intermediate states.

In Fig. **??**, I compare the solution returned by Path Length Rapidly-exploring Random Tree to the human guided trajectory on a single flight from the dataset. Path Length Rapidly-exploring Random Tree is given 3 minutes to find a solution. Each state in trajectory is plotted as a colored circle and adjacent circles are connected by a spline. For the human guided trajectory, I plot the waypoints provided by the flight tracking record. I project each trajectory onto the xy-plane to give a better perspective. In the human guided trajectory and Path Length Rapidly-exploring Random Tree each state is 30 seconds and 180 seconds apart, respectively. The maximum

edge length for Path Length Rapidly-exploring Random Tree is oversized so its solutions collapse to the Dubins curve.

When the maximum edge length is decreased, I notice that Path Length Rapidly-exploring Random Tree can no longer reach the goal reliably. Since the size of the goal region is small compared to the size of the state space, many samples explore unproductive regions and eat into runtime. The complexity of the non-holonomic constraints impose an additional penalty on the algorithm because more samples are required to complete paths which are feasible by an aircraft. Reaching the goal region is not a problem for Anytime Repairing A* variants, which indicates that the information provided by a heuristic may be important in exploring efficiently. Path Length Rapidly-exploring Random Tree does not have access to heuristic information which may contribute to the problem. Further, the results in Tab. **??** show that Path Length Rapidly-exploring Random Tree expands fewer states than Anytime Repairing A* within the same runtime. This means that Path Length Rapidly-exploring Random Tree does not explore the search space as throughly as Anytime Repairing A*. The smaller number of state expansions by Path Length Rapidly-exploring Random Tree is likely due to the overhead of computing the k-nearest neighbors because the Dubins curve needs to be calculated for each of the candidate neighbors.

In the motion cost variant of Rapidly-exploring Random Tree, I try to address the problem of goal reachability by informing the sampling procedure using the distance to the goal with the aim of increasing the efficiency of each expanded state. Unfortunately, even with a non-uniform sampling procedure, the Motion Cost Rapidly-exploring Random Tree variant does not find the goal region reliably. Tab. **??** shows that only 0.1% of Motion Cost Rapidly-exploring Random Tree trials return a solution. Motion Cost Rapidly-exploring Random Tree still expands fewer states than Motion Cost Anytime Repairing A* which suggests that the number of state expansions is still limiting the performance of Rapidly-exploring Random Tree. The lack of practical solutions from the Rapidly-exploring Random Tree variants make comparison with Anytime Repairing A* difficult. As such, my results suggest that the implementations of Rapidly-exploring Random Tree that I test here are insufficient for application to this planning problem.

Now, I consider the impact of the penalty factor on the solutions returned by Anytime Repairing A*. I find that the introduction of the penalty factor reduces the efficiency of the search. In the results shown in Tab. **??**, Motion Cost Anytime Repairing A* solves a smaller fraction of trials than Path Length Anytime Repairing A*, requires longer to find an initial solution, has more iterations, and expands more states. My result is different than the result obtained in [**?**] where Tolstaya et al. see an increase in the efficiency as their learning procedure converges. I think that this is likely due to the differences in learning the penalty factor. It could be the case that Motion Cost Anytime Repairing A* is attempting to minimize the penalty too much and is no longer guided by the heuristic. The introduction of a tuning parameter on the importance of the penalty may yield more efficient search with Motion Cost Anytime Repairing A*.

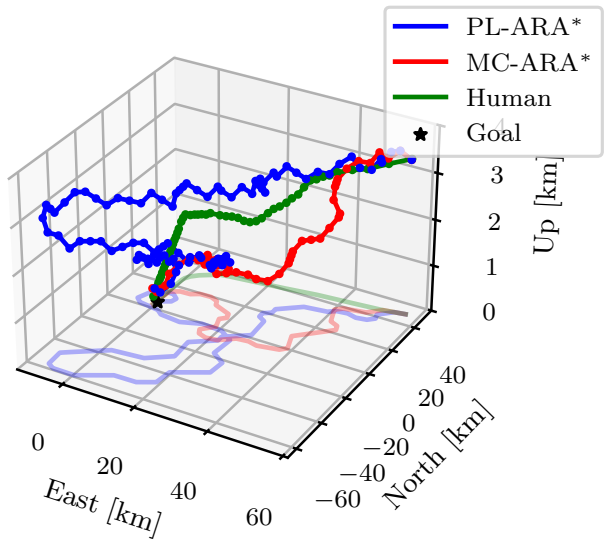

Fig. 5. A trajectory plot comparing a solution found by Path Length Anytime Repairing A*, Motion Cost Anytime Repairing A*, and the human guided reference. The goal region is denoted with a star. The plot axes are expressed using the East, North, Up convention with the origin at Seattle-Tacoma Airport.

Despite the inefficient search, I find that the solutions returned by Motion Cost Anytime Repairing A* follow the human guided trajectories more closely than Path Length Anytime Repairing A*. I can't make statistical claims about this result because I did not record the path length metric for solutions from Motion Cost Anytime Repairing A*. Instead, I present a heuristic case in Fig. **??** which shows that Motion Cost Anytime Repairing A* returns a shorter solution than Path Length Anytime Repairing A*. The figure is formulated using the same method as Fig. **??**. The Motion Cost Anytime Repairing A* trajectory mimics the descent pattern of the human trajectory whereas the Path Length Anytime Repairing A* trajectory descends very slowly in wide loops. The trajectory returned by Motion Cost Anytime Repairing A* is also much more smooth.

IX. CONCLUSION

In this work, I implement and compare graph-based and sampling-based planning algorithms for air traffic control using real flight trajectories from Seattle-Tacoma Airport. My results demonstrate that Anytime Repairing A* consistently outperforms Rapidly-exploring Random Tree in both finding solutions and achieving costs comparable to human-guided trajectories. The Anytime Repairing A* variants successfully solve 96.1% and 79.8% of trials for the path length and motion cost formulations, respectively, within 180 seconds. In contrast, Rapidly-exploring Random Tree variants struggle to reach the goal region reliably, with Motion Cost Rapidly-exploring Random Tree solving only 0.1% of trials.

The poor performance of Rapidly-exploring Random Tree stems from two key factors. First, the non-holonomic con-

straints of the Dubins airplane model require careful trajectory planning, which increases the computational overhead of the k-nearest neighbors calculations needed for rewiring. Second, the small goal region relative to the large search space means that uniform and weakly-informed sampling procedures spend significant time exploring unproductive regions. These findings suggest that heuristic information may be essential for efficient exploration in planning problems with tight kinodynamic constraints and small goal regions.

The learned penalty function successfully encodes common routes from human-guided trajectories, as shown in Fig. **??**. However, the Motion Cost Anytime Repairing A* variant requires significantly more computation time than Path Length Anytime Repairing A* to find solutions, which indicates that the current penalty formulation may be too conservative or that the discretization is too fine. Future work should explore alternative penalty learning methods that better balance the trade-off between following human examples and maintaining computational efficiency.

Future work should explore several directions to improve the performance of automated air traffic control planning. First, investigating adaptive or learned heuristics for Rapidly-exploring Random Tree could help address the goal reachability problem by providing better guidance toward feasible solutions. Second, the penalty learning approach could be extended to incorporate temporal information, such as time-of-day patterns or weather conditions, which would better capture the dynamic nature of air traffic control decisions. Third, the current formulation assumes single-agent planning, but realistic air traffic control requires coordinating multiple aircraft simultaneously. Extending these methods to handle multi-agent scenarios with inter-airplane safety constraints would be an important step toward practical deployment. Finally, the discretization parameters for both the configuration space and penalty factor should be optimized through systematic parameter tuning, as the current choices significantly impact both solution quality and computational efficiency. Addressing these challenges would move automated air traffic control systems closer to practical implementation.

REFERENCES

[1] E. Tolstaya, A. Ribeiro, V. Kumar, and A. Kapoor, "Inverse optimal planning for air traffic control," in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2019, pp. 7535–7542.

[2] O. Tange, *GNU Parallel 2018*. Ole Tange, Apr. 2018. [Online]. Available: https://doi.org/10.5281/zenodo.1146014