

USING JET ROUTES TO MODEL PATH RE-ROUTING IN THE NATIONAL AIRSPACE SYSTEM

Brian A. Malloy*
Department of Computer Science
National University of Ireland
Maynooth, CO Kildare
Ireland
bmalloy@cs.may.ie

Dean E. Bushey
Ramstein Air Base
PSC 1, Box 768
APO AP 09009
Germany
dbushey@surf1.de

Shaowu Yang
Department of Computer Science
Clemson University
Clemson, SC 29634
USA
yang@cs.clemson.edu

KEYWORDS

Class Diagram, Object-oriented, Floyd's algorithm, Jet routes, Fix, Airport Fix, National Airspace System

Abstract

Air traffic congestion and management of air traffic flow is becoming an increasingly important problem for travel in the national airspace system (NAS). One technique for analyzing the growth of complex systems is simulation modeling. Previous models of the NAS have used a bucket approach where each sector in the NAS is modeled as a bucket; planes enter the bucket when congestion is below a predefined tolerance level. However, the NAS is partitioned into sectors with clearly defined paths that the majority of air traffic must negotiate. These paths, or jet routes, intersect at navigation points or fixes. In this paper, we present a technique for modeling jet routes and fixes for air traffic in the NAS. We use Floyd's algorithm to compute the path from a source airport to a destination airport. We begin by modeling normal air traffic and then extend the model to include weather fronts and congestion. Comparisons of our C++ simulator to previous work show that our jet route approach is more efficient yet provides a more detailed simulation.

1 INTRODUCTION

Air traffic congestion and management of air traffic flow is becoming an increasingly important problem. Inefficiencies in the National Airspace System (NAS) have cost commercial airlines operating in the United States in excess of \$3.5 billion per year(?). Prognoses indicate that demands and strains on the system will increase dramatically; U.S. Secretary of Transportation, Federico Pena, reported that there will be a 45% increase in air travel in the next ten years(?). The Federal Aviation Administration (FAA) estimates that the number of passengers traveling on U.S. airlines will grow from 580 million in 1995 to nearly 800 million by 2003(?). Much effort and

money is being allocated to address the problems that increased traffic will incur.

One technique for analyzing the growth of complex systems is simulation modeling, which has proven invaluable for measuring the impact of local changes on the overall system. These models highlight inefficiencies and bottlenecks, facilitating solutions to uncovered problems. Models for air traffic control are either fine-grained(?) or coarse grained(?, ?, ?). Fine-grained models focus on the detailed operation of a particular airport, permitting measurement of controller workload. Coarse-grained models focus on global airspace, permitting study of one of the critical variables in national air travel: airplane delay. The detail needed in fine-grained modeling has limited the approach to the study of a single airport; thus, the fine-grained approach is unsuitable for modeling traffic delay and path re-routing. The course-grained approach partitions the NAS into sectors or buckets. Planes wishing to fly through a bucket may do so if congestion in the bucket is within a predefined tolerance level; otherwise, planes must wait at the bucket boundary until congestion within the bucket is reduced. The bucket approach has been exploited to demonstrate the benefit of dynamically re-routing traffic around sectors with reduced capacity due to weather fronts or accidents(?).

The problem with the bucket approach is that it does not provide sufficient detail for model validation nor does it capture the true behavior of planes traversing the NAS. The airspace over the United States is partitioned into sectors with clearly defined paths through these sectors that the majority of traffic must navigate. These paths, or jet routes, intersect at navigation points or enroute fixes. Associated with each airport is an arrival fix that serves as an entry or exit point. Previous models of the NAS have not modeled jet routes or fixes.

In this paper, we present a technique for modeling jet routes and fixes for air traffic in the NAS. Planes in our model depart an airport, ascend to an airport fix and then travel along a jet route until the plane arrives at the destination airport fix and destination airport. Our jet routes intersect at enroute fixes where delays may occur due to congestion at the fix. We use Floyd's algorithm to compute the path from the source airport to the destination airport(?). The path is represented by

*Brian is currently on sabbatical from Clemson University

use our model to measure the effects of dynamically re-routing traffic around fixes with reduced capacity. The design of our model exploits object technology to produce a simulator that is extensible to modification and additional functionality. Comparisons of our C++ implementation to the simulator in reference(?) indicate that our jet route approach is not slowed by the increased detail provided by our model; moreover our simulator was, on average, more than twice as efficient as the bucket approach of reference (?).

The remainder of this paper is organized as follows. In the next section we provide background about air travel in the NAS and an overview of Floyd’s algorithm. In Section 3, we present our object-oriented model followed in Section 4 by a discussion of our implementation. We describe the results of our experiments in Section 5 and draw conclusions in Section 6.

2 BACKGROUND

In this section, we provide background about national air traffic. We begin with an overview of NAS, the National Airspace System. We then describe travel across the NAS including a sample flight from Raleigh-Durham to Atlanta with a corresponding flight plan.

2.1 Overview of the National Airspace System (NAS)

The airspace over the United States, or the *National Airspace System* (NAS), includes the continental US and its international arrival and departure airspace. The NAS is partitioned into sectors, or 3 dimensional areas of non-overlapping airspace with clearly defined paths through these sectors that the majority of the air traffic must navigate. These paths, or *jet routes*, intersect at navigation points, or *enroute fixes*. Associated with each runway is an *arrival fix* and a *departure fix* that serve as entry or exit points. Figure 1 contains an *Enroute chart sample* that illustrates portions of the Jacksonville, Washington and Atlanta Air Route Traffic Control Centers (ARTCC). The small concentric circles represent military fields; for example, Dobbins AFB is illustrated in the lower left corner of Figure 1. The larger circles, which resembles a compass face, represent navigational fixes; for example, the fix for Atlanta is illustrated in the lower left corner of Figure 1. Below the fix for Atlanta is an information box containing the fix name, the navigational radio frequency for direction (VOR), a three letter identifier for the fix, and a navigational radio frequency for both direction and distance (TACAN). The black lines between enroute fixes are the jet routes, and each route contains one or more designator; for example, the jet route between Atlanta and Spartanburg is designated J14-37, which indicates that the route can be designated as J14 or J37.

proceeds to the corresponding departure fix servicing that runway. If there is a conflict at the departure fix or in the departure sequence, then the aircraft must hold at the airport awaiting takeoff clearance, causing a delay. After departure, the aircraft waits at the fix until the air traffic controller gives clearance into the enroute sector. After receiving clearance, the aircraft enters the first sector, the one that contains the departure runway.

Once cleared into the enroute sector structure, the aircraft proceeds across sectors to its final sector containing the destination runway for the flight. Each time the aircraft nears a sector boundary, it must receive clearance from the air traffic controller before entering the sector. If there is a conflict in the new sector, the aircraft will either hold in its present sector, causing a delay, or be rerouted to a neighboring sector. Upon reaching the last sector in a flight, the plane must approach a runway. This involves going from the high enroute structure, down to the destination runway terminal area. As in departures, there is an arrival fix that channels traffic to each runway. If there is a conflict, the aircraft will be issued hold instructions, causing delay. If there are no conflicts, the aircraft receives an approach clearance and lands at the runway. At each fix in this flight, the aircraft could be forced to hold, causing costly delays.

```

algorithm   floyd
input      D: matrix containing distance between fixes connected by an e
output     D: matrix containing shortest distance between fixes
output     P: matrix containing paths between fixes

begin floyd
  for int k = 0; k < MAX; ++k)
    for int i = 0; i < MAX; ++i)
      for int j = 0; j < MAX; ++j)
        if (D[i][k] + D[k][j] < D[i][j]) {
          D[i][j] = D[i][k] + D[k][j];
          P[i][j] = k;
        }
  end floyd

```

Figure 2: **Floyd’s algorithm.** This figure lists the algorithm for computing the distance between vertices and the path between those vertices. Upon termination of the algorithm, $D[i][j]$ will contain the shortest distance from vertex i to vertex j . The path from vertex i to vertex j is found by backtracking from $P[i][j]$.

2.3 Floyd’s Algorithm

To compute the distance and path from one airport in the NAS to another, we use the algorithm illustrated in Figure 2(?). As the figure shows, the algorithm exploits two matrices and three nested **for** loops to compute shortest distance and path information for a graph $G(V, E)$, where V is the set of vertices and E is the set of directed edges. Matrix D is an adjacency matrix where each entry $D[i][j]$ is initialized to the distance between the i^{th} and j^{th} vertex if i and j are connected by an edge, otherwise $D[i][j]$ is initialized to ∞ . Matrix P is a path matrix where each entry $P[i][j]$ is given some initial value, for

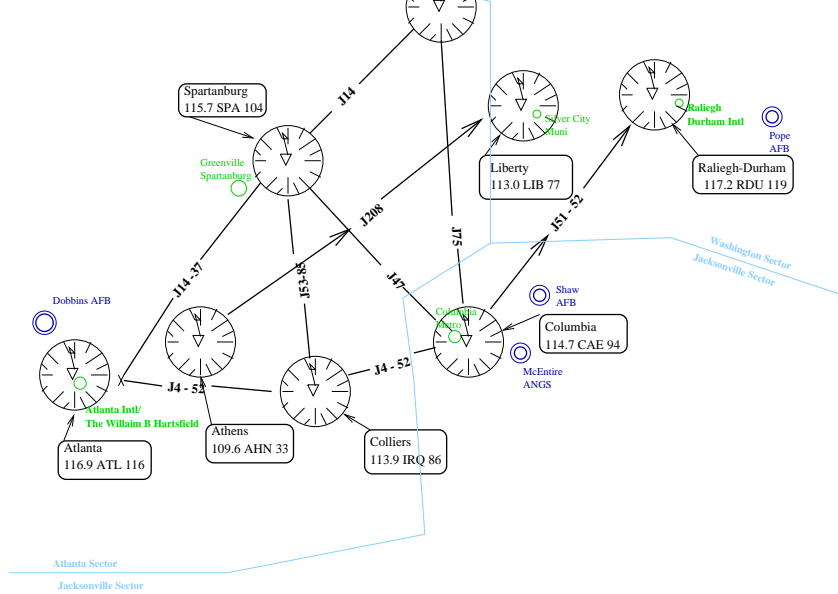


Figure 1: *Enroute chart sample*. This is part of the Southeast Enroute High Altitude Chart. The compass faces represent navigation fixes, oriented to magnetic north, with the arrow pointing to true north. The information box with each navigational fix denotes the fix name, the VOR directional frequency, the identifier, and the TACAN vector frequency. The smallest single circles show civilian airfields, and the concentric circles show military fields. The dark lines between fixes are the jet routes. The light lines show sector boundaries.

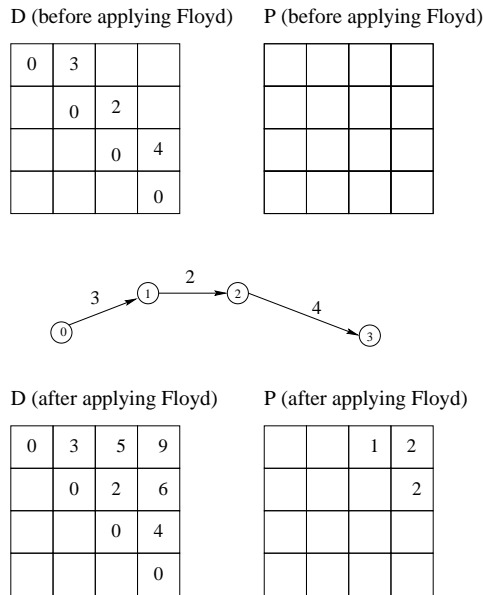


Figure 3: *Sample use of Floyd's algorithm*. This figure illustrates the adjacency matrices that we use with Floyd's algorithm to compute the shortest distance, $D[i][j]$, between vertices i and j in a graph, and the matrix P that stores the shortest path. The top of the figure illustrates the initial values of D and P ; unlisted values of D are ∞ and unlisted values of P are -1 . The middle of the figure illustrates four high fixes and their adjoining edges. The bottom of the figure illustrates final values of D and P .

example -1. After using Floyd's algorithm, $D[i][j]$ will contain the shortest distance between the i^{th} and j^{th} ver-

tex and $P[i][j]$ will contain the vertex k that led Floyd to find the smallest value of $D[i][j]$. The nested loops in Figure 2 make n iterations over each vertex in the graph so that after the k^{th} iteration, $D[i][j]$ will contain the smallest length of any path from vertex i to vertex j that does not pass through a vertex numbered higher than k .

Figure 3 illustrates a sample use of Floyd's algorithm. The top of Figure 3 illustrates matrices D and P before using Floyd's algorithm, the middle of the figure illustrates a sample graph with four vertices and the bottom of the figure illustrates matrices D and P after using Floyd's algorithm. For the graph in Figure 3, vertices 0 and 1 are connected by an edge of length 3, vertices 1 and 2 are connected by an edge of length 2, and vertices 3 and 4 are connected by an edge of length 4 units; these connections are illustrated in matrix D where locations $D[0][1]$, $D[1][2]$ and $D[2][3]$ have values 3, 2, and 4 respectively. The diagonals of the matrix are initialized to zero to indicate that the distance from a vertex to itself is zero. The figure illustrates that, since vertices 1 and 3 are not directly connected, the initial value of $D[1][3]$, before using Floyd's algorithm, is ∞ . Similarly the path matrix P has all of its values initially set to -1 .

After using Floyd's algorithm, the value of $D[0][2]$ is 5 to indicate that the distance from vertex 0 to vertex 2 is 5 units. To see how Floyd updates the D and P matrices, consider the **if** statement in Figure 2 when the loop control variables i , j and k , have values 0, 2, and 1 respectively; the **if** condition will evaluate to *true* since $D[0][1] + D[1][2]$ is 5 and $D[0][2]$ is ∞ . The statements inside the **if** statement will execute so that $D[0][2]$ is updated to 5 and $P[0][2]$ is updated to 1. The updated value

2 after using Floyd's algorithm. The shortest path from vertex 0 to vertex 3 can be reconstructed by stacking the values of $P[0][3]$ and $P[0][2]$; thus, the shortest path from vertex 0 to vertex 3 is 0, 1, 2, 3. The interested reader can find more information about Floyd's algorithm in reference (?).

3 THE MODEL

In this section, we describe the model that captures our simulation of jet routes in the NAS. We begin by overviewing five objects that describe the network of jet routes in the NAS: *Network*, *Airplane*, *Airport*, *Airport Fix* and *High Fix*. We then overview the nine events in our model including a discussion of the take off event, `takeOffEvent`.

3.1 Overview

We model air traffic control as a discrete event-driven simulation. Objects in our model include airplanes, airports, airport fixes, high fixes and weather fronts. In our model, an *airport fix* differs from a *high fix* in two important ways: (1) an *airport fix* has a higher capacity than a high fix, and (2) each airport is associated with one or more *airport fixes* and any plane departing the airport must enter one of the associated airport fixes.

Since our simulation is event-driven, we use a priority queue of events, with the time stamp of each event setting the priority. The events in the simulation are associated with each of the objects and they include events for airplane takeoff and landing, airplanes entering and exiting airport and high fixes, weather events and a special event to terminate the event-driven simulation. Airplanes are generated at airports and interact with other objects during the simulation. We now summarize the actions of five objects in our simulation including the *Network*, *Airport*, *Airport Fix*, *Airplane* and *High Fix* object.

1. **The Network Object** The Network object is the most important object in our model; in a manner of speaking, the network *is* our simulation. This object includes data structures to describe the jet routes in the NAS, as well as data structures to contain airports, airport fixes and high fixes. It is the Network object that schedules take off events (`takeOffEvent`).
2. **The Airport Object** The airport object generates airplanes and takeoff events, and terminates a flight when an airplane lands. The airport also passes messages to existing planes about takeoff conditions and delays. The size of an airport is represented by the number of airplanes that an airport can contain; a takeoff event is generated for each airplane in the airport. If the number of airplanes at the airport exceeds its size, then a landing airplane is delayed. Airports maintain statistics about numbers of takeoffs, landings and takeoff delay.

capacity than a high fix. An airport fix allows planes to enter and exit and maintains statistics such as delay time due to a crowded airport or a weather front.

4. **The Airplane Object** When an airplane object is created, its destination airport is randomly selected, the path to the destination airport is computed using Floyd's path finding algorithm and the path is stored in the airplane object. Each airplane maintains its source and destination airport, path to the destination together with information about its type, speed, and delay along the path. When the airplane arrives at its destination, statistics are gathered and, in our simulation, the airplane object is deleted.
5. **The High Fix Object** A high fix object manages planes entering, traversing and exiting the high fix. We model a high fix as a circle and the time to travel through the high fix is the ratio of the diameter of the high fix and the speed of the airplane. An airplane is delayed if it attempts to enter a high fix that has reached its maximum capacity. Without re-routing, airplanes must wait at a high fix that is experiencing a weather front or whose capacity is reached. With re-routing, planes may radio ahead and, if the fix is crowded, the plane may re-route around the crowded fix.

3.2 Static Description of the Model

The classes to instantiate the objects described in the previous section are illustrated in Figure 4. The squares in the figure represent classes, with the name of each class listed at the top of the square and the methods and data attributes listed in the middle and bottom of the square; we list only the methods and data attributes that are important for discussions that follow. The lines connecting squares indicate the association between classes and lines that include a triangle indicate an inheritance relationship. The right side of the figure shows a circle, **Event Class Framework**; this circle represents our class framework for events and will be discussed in a latter section.

The left side of the figure shows two inheritance frameworks with base classes `Fix` and `Simulation`. `Fix` is an abstract base class (ABC) that includes subclasses `ArptFix` and `HighFix`, whose instances represent airport fixes and high fixes respectively. The virtual abstract functions in `Fix` guarantee that each subclass will include functionality to test for fullness, `full()`, print statistics, `showStats()`, and return the unique number identifying each fix, `getFixId()`. Subclass `HighFix` contains a vector with an entry for each connected fix.

The base class `Simulation`, illustrated in the middle of Figure 4, includes functionality to run the simulation, `run()`, schedule events, `scheduleEvent()` and terminate the simulation, `endSimulation()`. Also included in `Simulation` is a data attribute for maintaining the

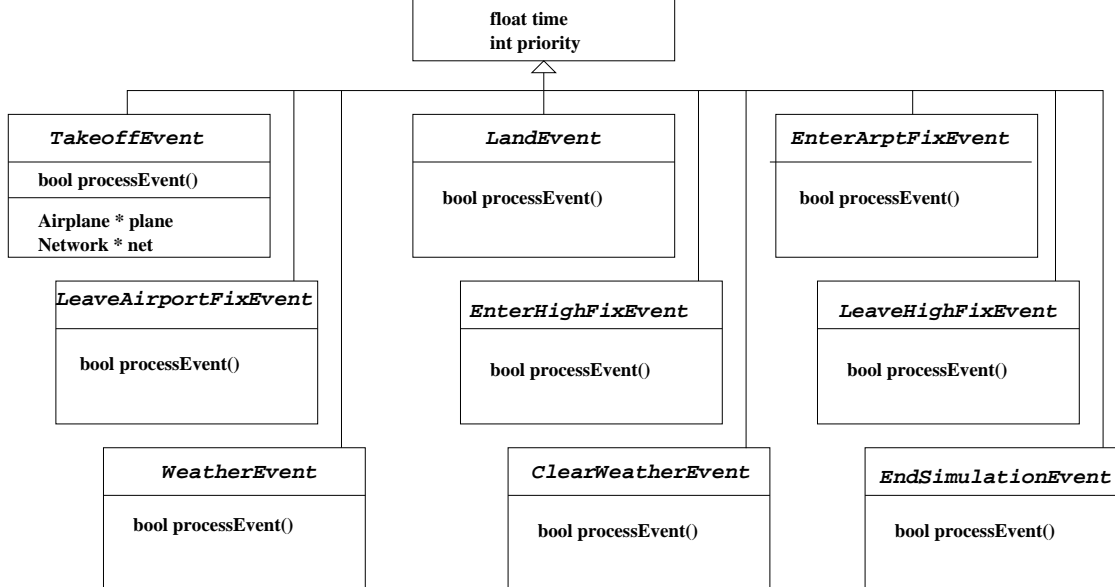


Figure 5: *Event Class Framework*. This diagram illustrates the nine events in the *Event Class Framework* of our simulation model.

Sample Input Data for Airports and Airport Fixes						
<i>Airport name</i>	<i>Type</i>	<i>Frequency of arrivals</i>	<i>No. of Runways</i>	<i>Plane capacity</i>	<i>Planes at start</i>	<i>Distance to fix</i>
Seattle	commercial	1.008	1	60	50	1
<i>Airport fix name</i>		<i>Capacity</i>		<i>Diameter</i>		<i>Distance</i>
Seattle		85		1		1

Table 1: *Input data for airports and airport fixes*. The data in this table are the parameters to our model that we used to describe sector 8. There are 50 sectors in our model and each sector is described in a similar fashion; each sector in our model has a unique description.

Table 1 illustrates the information that is read by the network object in step (2) to describe the *Seattle* airport and its airport fix. The top two rows of the table lists the *Airport name*, *Seattle*, the *Type* of traffic that the airport supports, commercial, the number of runways, 1, the number of planes that can be in the airport simultaneously, 60 planes, the number of planes at the beginning of the simulation, 50 planes, and the distance in nautical miles to the airport fix, 1 mile. The bottom two rows of Table 1 lists the name of the associated airport fix, *Seattle* and the capacity of the airport fix, 85, which is the number of airplanes that can simultaneously be in the airport fix. Also listed in the bottom two rows is the distance across the airport fix, 1 nautical mile, and the distance from the airport to the airport fix.

Table 2 illustrates the information required to describe a high fix, *Tacoma Narrows* and lists the three fixes connected to *Tacoma Narrows*. The top of Table 2 lists the capacity of of the high fix, 5, which means that 5 planes that can be in the fix simultaneously. Also listed at the top of the table is the diameter of the high fix and the number of routes connected to the fix. The bottom rows of Table 2 lists the three high fixes connected to *Tacoma*

Narrows as well as their distances.

4.2 Generating Airplanes

The class *Airplane*, whose instances are the airplanes in our simulation, is illustrated in Figure 4 of Section 3.2. As the figure shows, airplane objects carry their path as well as the distances between fixes along the path; these objects also include pointers to both the take off airport and the destination airport. The algorithm to generate an airplane is a member of the *Network* class. Input to the algorithm is the unique id for the source airport and the output is the newly created instance of *Airplane*.

The steps required to generate an airplane include initialization of the time the plane leaves the airport, *takeoff_time*, the destination airport, *dest_airport*, a pointer to the airport fix associated with the source airport, *takeoff_fix*, and a pointer to the airport fix associated with the destination airport, *landing_fix*. The path to the destination airport and distances between fixes along the path are computed and stored in *path_vector* and *distance_vector*. After these initializations are complete, as the algorithm in the fig-

5 EXPERIMENTS

In this section, we use our simulator to show that a weather front can contribute heavily to traffic delay in the NAS. We show that by using dynamic re-routing around a weather front, we can considerably reduce delay in the system. We also demonstrate efficiency by comparing the execution time of our simulator, *Jet Routes*, with the execution time of a previous simulator, *MATS*(?). To compare the two models, we adjust the parameters to *Jet Routes* so that the number of takeoffs, landings and airports match those of *MATS*. Also, both *Jet Routes* and *MATS* model a 24 hour period with fewer planes traveling during the early hours, or morning hours, of the simulated time period. All of the experiments in this section were executed 12 times with the highest and lowest values discarded; the numbers reported are averages over 10 executions. To formulate a valid comparison with *MATS*, all experiments were executed on a Gateway personal computer running at 133 MHz operating under Solaris.

The statistics illustrated in Table 3 were gathered for our simulator, *Jet Routes*, modeling 50 airports, 50 airport fixes and 100 high fixes. The three rows of data in Table 3 list statistics for three experiments. The first experiment, illustrated in the first row of the table, serves as a control with no weather front causing delay at any high fixes, airports or airport fixes. The second experiment, illustrated in the second row of the table, used the same parameters as the first experiment except that a weather front was introduced at the Lakeview high fix; planes whose path included the Lakeview high fix were delayed at the fix and were not re-routed. The third experiment, illustrated in the third row of the table, used the same parameters as the second experiment except that planes were re-routed around the Lakeview high fix.

The first two columns of data in Table 3 show that for all three experiments the number of takeoffs and landings remained approximately the same. For example, there were 51,483 takeoffs in the first experiment 51,007 takeoffs in the second experiment and 51,228 takeoffs in the third experiment. The first two columns of Table 3 also show that there were more takeoffs than landings in our simulation since the simulator terminated after modeling a 24 hour period, leaving some planes airborne. The third and fourth column of Table 3 illustrates the delay induced by the weather front over Lakeview. In the first experiment, the third column lists 56 planes delayed and the fourth column lists 112 minutes delay; in the second experiment, the third column lists 1,194 planes delayed and the fourth column lists 2,388 minutes delay; thus, the weather front over Lakeview increased the delay in the system by an order of magnitude. In the third experiment, the third column lists 135 planes delayed and the fourth column lists 270 minutes delay; thus, re-routing around the weather front reduced the delay almost to that of the first experiment. The fifth column in the table

The sixth column explains the increase in execution time for the three experiments since the number of events to be processed increased from 799,148 events in the first experiment to 816,192 events for the third experiment; thus, re-routing around the weather front increased the number of events in the simulator.

To show the efficiency of our simulator, Table 4 compares statistics for our simulator, *Jet Routes*, with those of a previous simulator, *MATS*(?). The three rows in the table list statistics for the three experiments referenced in Table 3. The first two columns of Table 4 show that approximately the same number of plane takeoffs occurred for both *MATS* and *Jet Routes* for the three experiments. For example, in the first experiment there were 50,091 plane takeoffs for *MATS* and 51,483 plane takeoffs for *Jet Routes*. The third and fourth columns show that approximately the same number of plane landings occurred for both *MATS* and *Jet Routes* for the three experiments. For example, in the first experiment there were 42,314 plane landings for *MATS* and 47,383 plane landings for *Jet Routes*. The fifth and sixth columns of Table 4 show that our simulator, *Jet Routes*, captures more detail in the simulation than *MATS*. For example, in the first experiment, our simulator processed 799,148 events while the *MATS* simulator only processed 282,409; thus, our simulator processed almost three times as many events as the *MATS*.

The final two columns of Table 4 illustrate that our simulator was more than twice as efficient as *MATS*. For example, for the first experiment, the *MATS* simulator required 165.75 seconds to process an average of 282,409 events while our simulator, *Jet Routes*, required only 61.25 seconds to process an average of 799,148 events. The path computation algorithm that we use in *Jet Routes* is efficient, running in linear time on average. Also, we compute the path at the beginning of each flight and possibly re-compute the path to circumnavigate a weather front. However, the *MATS* simulator does not precompute the path but rather chooses a sector at each step of the simulation. Thus, our *Jet Routes* approach produces both a more detailed and more efficient simulator than the *MATS* simulator of reference (?).

6 CONCLUDING REMARKS

We have presented an object-oriented design and implementation of a simulator for jet routes in the National Airspace System (NAS). Our design presentation includes class diagrams for the inheritance frameworks as well as a framework for the events in the simulation. Our C++ implementation exploits the standard library (STL) for all of the important containers, including the priority queue of events, together with standard library algorithms and adapters. We use an algorithm from Floyd(?) to compute the shortest path along the jet routes. Our implementation is more than twice as efficient as a previous simulator that uses a bucket approach and we show that re-routing around weather fronts can decrease delay

Weather	51,007	46,125	1,194	2,388	65.67	801,343
Re-routing	51,228	46,932	135	270	69.96	816,912

Table 3: *Summary table.* This table illustrates the statistics gathered during three experiments captured in each of the three rows of data. In the first row of the table, the high fixes are large enough to handle traffic with minimum delay. In the second row, the high fix over Lakeview experiences a weather front that reduces the capacity of the fix; no re-routing is used for the experiments in the second row. In the third row, the high fix over Lakeview experiences a weather front and traffic is re-routed around that fix while the weather front is in effect.

Simulation Parameters	Performance Comparison							
	<i>Total takeoffs</i>		<i>Total landings</i>		<i>Total events</i>		<i>Execution time (sec)</i>	
	MATS	Jet Rts	MATS	Jet Rts	MATS	Jet Rts	MATS	Jet Rts
(Control)	50,091	51,483	42,314	47,383	282,409	799,148	165.75	61.25
Weather	49,634	51,007	44,257	46,125	289,935	801,343	169.39	65.67
Re-routing	50,454	51,228	44,974	46,932	285,003	816,912	168.62	69.96

Table 4: **Comparison of statistics**

cycles by an order of magnitude. Our continuing work on the simulator includes an extension to use weighted edges to precompute a path through the NAS that avoids pre-announced weather fronts.

ACKNOWLEDGEMENT

The authors would like to thank the referees; their comments greatly facilitated the revision of this paper.

References