Development of Optimization and Simulation Models for the Analysis of Airfield Operations

by

Hojong Baik

Dissertation submitted to the faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Civil Engineering

Antonio A. Trani, Co-Chair Hanif D. Sherali, Co-Chair

Dusan Teodorovic

Hesham Rakha

Julio C. Matinez

May 2000

Blacksburg, Virginia

Keyword: Integer Programming, Microscopic Simulation Model, Airport Terminal Area

Development of Optimization and Simulation Models for the Analysis of Airfield Operations

by

Hojong Baik

Co-Chairmen: Professor Antonio A. Trani, and Professor Hanif D. Sherali.

(ABSTRACT)

This research is concerned with the modeling and development of algorithmic approaches for solving airport operational problems that arise in Air Traffic Control (ATC) systems within the terminal area at hub airports. Specifically, the problems addressed include the Aircraft Sequencing Problem (ASP) for runway operations, the Network Assignment Problem (NAP) for taxiway operations, and a simulation model for the evaluation of current or proposed ATC system in detail.

For the ASP, we develop a mathematical model and apply the Reformulation-Linearization-Technique (RLT) of Sherali and Adams to construct an enhanced tightened version of the proposed model. Since ASP is NP-Hard and in fact, it is a variation of the well-known Traveling Salesman Problem with time-windows, sub-optimal solutions are usually derived to accommodate the real-time constraints of ATC systems. Nevertheless, we exhibit a significant advancement in this challenging class of problem. Also for the purpose of solving relatively large sized problems in practice, we develop and test suitable heuristic procedures.

For the NAP, we propose a quasi-dynamic assignment scheme which is based on the incremental assignment technique. This quasi-dynamic assignment method assumes that the current aircraft route is influenced only by the previous aircraft assigned to the network. This simplified assumption obviates the need for iterative rerouting procedures to reach a pure equilibrium state which might not be achievable in practical taxiway operations. To evaluate the overall system, we develop a microscopic simulation model. The simulation model is designed to have the capability for reproducing not only the dynamic behavior of aircraft, but also incorporates communication activities between controllers and pilots. These activities are critical in ATC operations, and in some instances, might limit the capacity of the facility.

Finally, using the developed simulation model named Virginia Tech Airport Simulation Model (VTASM) in concert with ASP and NAP, we compare the overall efficiencies of several control strategies, including that of the existing control system as well as of the proposed advanced control system.

This dissertation is dedicated to the memory of
my mother ***Gyu-young Ahn***

Acknowledgements

I would first like to thank my advisors, Dr. Antonio A. Trani and Hanif D. Sherali, for their invaluable guidance, patience and inspiration during my entire studying at Virginia Tech. Without their supports, this dissertation would not have been completed. I also indebted to them for providing financial support through the National Center of Excellence for Aviation Operations Research (NEXSTOR) and for numerous opportunities they have presented me with.

I have also been honored to have Dr. Dusan Teodorovic, Dr. Hesham Rakha, and Dr. Julio Matinez as members of my committee. Special thanks go to my previous teacher, Dr. Kangwon Lim, at Seoul National University for his encouragement.

I would like to thank my friends including Dr. Youn-soo Kang, Myung-hyun Lee, Dr. Jeong-gyu Kang, Dr. Gyuhae Park, Heunggweon Sin, Sungpil Hahm, and Kyungho Ahn for their support and warmhearted friendship.

I could not thank enough my parents, Chang-ki Baik and Gyu-young Ahn, and parents-in-law, Won-woo Park and Eun-ja Kim, for their unconditional love and faith in me. Thanks also go to my sister's, brother's and sisters-in-law's families. My special thanks go to my wife's aunt M.D. Soo J. Kim for her constant encouragement and assistance to my family.

I would like to express my deepest gratitude to my beloved wife, Sunghye Park, and my daughter, Julia Inhye Baik, for their support, patience, understanding, and endless love throughout my graduate studies in Blacksburg.

Contents

List of Figures

List of Tables

Chapter 1. Introduction

1.1 Motivation and Purpose

According to Federal Aviation Administration (FAA) statistics, there were 27 airports in U.S. having more than 20,000 hours of total annual delay in 1997. These delays represent a cost exceeding 2.4 billion dollars. By the year 2003, the number of airports having annual delays exceeding 20,000 hours could be 31, unless capacity improvements are made [FAA, 1998].

Efforts to augment current capacity –by building new airports, or expanding existing ones- are expensive, time consuming, and environmentally controversial. On the other hand, proposals to change the demand for the Air Traffic Control (ATC) operations by, for example, imposing higher landing fees can lead to legal proceedings. Hence there is great interest in using existing capacity more efficiently by improving air traffic control procedures.

The critical bottleneck in an ATC system is the capacity inside a radius of about 50 nautical miles (nm) around an airport (i.e., airport terminal area). There are three types of air traffic control activities in this terminal area: ***aircraft sequencing operations*** which control the traffic inside the airport terminal area, ***runway operations*** which control aircraft landings and departures, and ***taxiway operations*** which guide aircraft from the gate to the runway for departures or vice versa for arrivals. For these activities, three types of air traffic controllers are involved: Final approach controllers who are responsible for the aircraft sequencing operations, local controllers who are responsible for runway operations, and ground controllers who are responsible for taxiway operations.

To achieve more efficient operations, it is desirable that those three operational components be coordinated as one task, rather than be considered as three separate ones. The purpose of this research project is to design an efficient method to model aircraft operations around the airport terminal area using an open system architecture. It is believed that this operational scheme could help alleviate the congestion around the airport terminal area, that is frequently observed at major airports in the United States and abroad. The underlying philosophy of this research project is reflected by Dear's remark [Dear, 1976]:

> "By utilizing today's computer power and instrument sophistication, the controller workload can be reduced and system performance improved through the use of computer-assisted decision-making, without affecting the controller's autonomy."

1.2 Problem Description

This dissertation focuses on the operation of aircraft inside the airport terminal area (or airfield) which is a 3-dimensional air space usually having 50 nautical miles of radius around airport runways. Once an arriving aircraft passes an entry point of airport terminal area, it approaches towards a runway. For the convenience of air traffic control, all arriving aircraft are required to pass the final approach fix (FAF) just before it touch down. The airspace between the FAF and runway is called final approach path (FAP) or final. In the FAP, aircraft are controlled to be spaced in certain amount of distance based on the minimum separation rules enforced by FAA. The minimum separation between arriving aircraft depends on the aircraft types involved. This separation rule is designed to minimize the leading aircraft's vortex effect to the following aircraft. Generally speaking, the separation is bigger when an aircraft follows a larger one than when it follows a smaller one. Once the arriving aircraft touchdowns and exits from the runway, it start to taxi to the gate.

On the ground, aircraft parking at gates load and unload either passengers or freight. After completing the loading and unloading processes, the aircraft begins to taxi to the runway along the designated taxiing path. Once the aircraft arrives at the end of runway, it starts to roll for takeoff. Figure 1.1 illustrates the configuration of airport terminal area.

Fundamentally speaking, the delays in any transportation facility occur when the demand for services exceeds its capacity. This basic rule can be applied to facilities around the airport terminal area, such as runways, taxiways and gates. For example, when there are more aircraft moving on a taxiway network than its capacity, some of taxiing aircraft would experience certain amounts of delay due to network congestion.

In this dissertation, we consider an airport terminal area as a combination of various resources having limited capacities. In our analysis, ***runway***, ***taxiway*** and ***communication channels*** are considered and accordingly this research focuses mainly on the analysis of air traffic operations on the airfield facilities. (Aspects related to the gate facilities are not included in research, but could be the subject of future studies.)

In short, this dissertation deals with the following three sub-problems: the ***Aircraft Sequencing Problem*** (**ASP**) to model runway operations, the ***Network Assignment Problem*** (**NAP**) to model taxiway operations, and the ***Simulation Model*** (**SM**) for addressing communication activities.

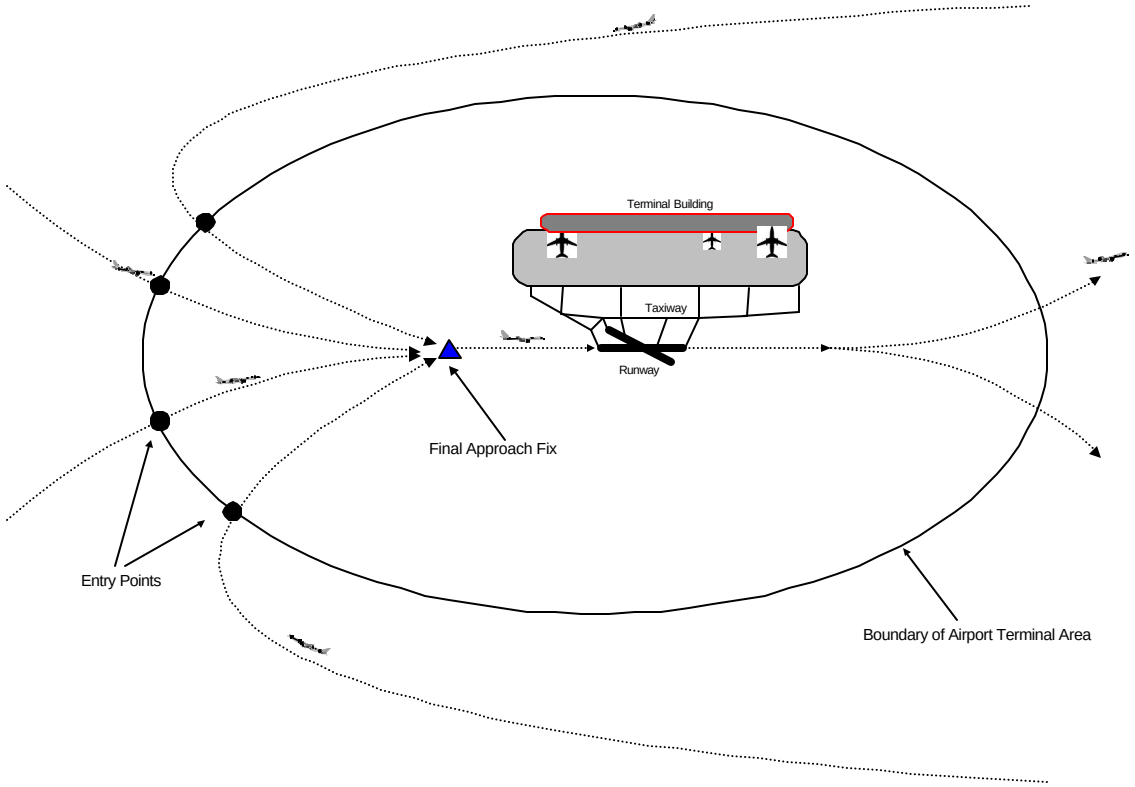


Figure 1.1 Configuration of Airport Terminal Area

The Aircraft Sequencing Problem (ASP) handles the runway operations and is developed under the guiding principle that if the arrivals and departures are sequenced intelligently, then an enhanced system efficiency will accrue. The ASP problem considers minimum separation rules enforced by FAA to protect consecutive aircraft from the dangers of wake-vortex effects.

To illustrate this point, note that during the final approach, air traffic controllers try to maintain a minimum of 5 nm when a heavy aircraft leads a small one. This distance is equivalent to about 196 seconds in time. If this sequence is switched so that the small aircraft leads the heavy one, the required distance between the two aircraft reduces to 3 nm, or about 75 seconds. Thus a saving of more than 100 seconds could occur by switching the sequence, assuming that this process is feasible to the individual aircraft time-window restrictions, and does not appreciably affect delays.

Previous studies conducted on this topic have mainly focused on the sequencing problem for arriving aircraft. Since many airports in the U.S. have dependent arrival and departure stream operations, consideration is given here to both these operations simultaneously. This makes the

problem more realistic, yet at the same time, more challenging. In order to manage aircraft traffic in congested airport terminal areas, an aircraft sequencing procedure can be used in conjunction with advanced Air Traffic Control (ATC) automation tools. Problem ASP can be stated more succinctly as follows:

Given *a set of aircraft data involving both arrivals and departures, including aircraft type, original* (*nominal or desired*) *schedule, a maximum delay time to be absorbed by each aircraft, minimum separation rules among aircraft, and the runway occupancy time* (*ROT*) *for each aircraft type.*

Find *an optimal sequence of aircraft which minimizes the total operation time/cost to finish all operations while satisfying both the minimum separation rule and the maximum delay constraints.*

Given the time recommendations prescribed by ASP, which include the exit times from the runway and the takeoff times at the runway, the next step is to solve Network Assignment Problem (NAP) to decide the taxiing route for all aircraft so as to minimize the congestion on the taxiway network. The NAP which deals with taxiing operations is defined as follows.

Given *a network configuration* (*a directed graph G*(*N,A*)) *of runways and taxiways, including a set of origin nodes, O; and a set of destination nodes, D; and time-dependent taxiing demands from each origin to each destination for a certain period. Also, given are prescribed landing and takeoff times at the runway for the aircraft, as obtained from model ASP.*

Find *a set of optimal routes for the departing aircraft to lead them from the gate to the departure queue, and for the arriving aircraft to lead them from the runway exit to the gate, in order to minimize total travel or fuel costs.*

In this analysis, we adopt the results of ASP as initial conditions for solving the NAP problem. In regular airport operations, landing aircraft have higher priority over departing and taxiing aircraft. This is because arrivals are both costly and safety-critical from a fuel consumption and operational point of view. For the same reason, the runway operation has a priority over ground taxiing. Hence, in the case where a conflict arises between an aircraft crossing a runway during Taxiing and an aircraft departing or landing on the runway, the latter has a priority. Based on view, we solve the ASP first and obtain the optimal runway operational sequence. We then solve

the NAP using the ASP optimal sequence. Figures 1.1 and 1.2 illustrate the relationship between the ASP and NAP problems for a single data horizon, including the required data and information flows.

At many busy airports, it is frequently observed that flights are delayed by communication congestion (due to controller workload). For instance, even after a departing flight is ready to taxi, the flight would be delayed at the gate until it obtains a taxiing clearance from the ground controller. In previous analyses for ASP and NAP problems, no consideration has been given to the communication process between controllers and pilots, although this is an important source of delay. Indeed, there is a distinct possibility that due to the communication delays, the prescribed ASP sequence and the suggested taxiing route obtained from NAP might not be achievable for some flights.

In order to portray the pilot-controller communication process, a ***Simulation Model*** (**SM**) is developed in this research. A continuous, microscopic simulation model provides very detailed information about the dynamic status of all relevant entities, i.e., flights, local controllers, and ground controllers. The dynamic status includes not only controllers' communication status but also aircraft dynamic behaviors such as speed, acceleration, position, etc., at every time interval of simulation. These microscopic features are embedded in the **Virginia Tech Airport Simulation Model** (**VTASM**), and distinguish separate this model from existing air traffic simulation models such as SIMMOD, TAMMS, etc., which employ disctere-event simulation instead. VTASM includes airport facilities such as runways, taxiways, and gates. (Runway thresholds play the role of a souce node for arriving flights and a sink node for departing flights.) The main tasks in the development of VTASM can be described as follows:

Given *a network configuration* (*a directed graph* $G(N,A)$) *of runways and taxiways, prescribed landing and takeoff times at the runway for the aircraft, as obtained from model ASP, and a set of optimal routes for both the departing and arriving aircraft from NAP.*

Find*, considering communication activities and aircraft dynamic movements, the aircraft status such as speed, acceleration, position, etc., and delays on the network.*

1.3. Organization of this Document

This dissertation is organized into six chapters. Chapter 2 is devoted to review previous studies pertinent to ASP problem. The proposed ASP is a version of the *asymmetric traveling salesman*

problem with time-windows (ASP-TW), and is NP-complete in terms of computational complexity. Because of this complexity, we develop two approaches for this problem, respectively geared toward finding an exact solution and an approximate solution via a heuristic method. In Chapter 3, the formulation of Problem ASP and the development of an effective approach to solve this problem are presented along with some computational results for both the exact and heuristic approach.

The NAP problem on the taxiway network is discussed in Chapter 4. We focus on proposing a network assignment strategy which is appropriate for our case. Even though the taxiway network is somewhat different from urban transportation networks, it is still helpful to review the state-of-art of both static and dynamic network assignment methods applied to urban transportation planning processes. Various types of shortest path algorithms which play an important role in NAP are also discussed along with their associated data structures. Two types of network assignment strategies are considered in this research.

In Chapter 5, we present procedures relevant to the development of a continuous, microscopic simulation model. The details of each component of the simulation model are provided in the context of an Object-Oriented Design (OOD) and an Object-Oriented Programming (OOP) approach. Computational results and a verification of the simulation model are also presented in this chapter.

Chapter 6 provides results for a case study using hypothetical flight schedules at the Reagan National Airport (DCA). A total of 12 scenarios are studied for several sequencing, network assignment and pilot-controller communication methods. Each scenario is tested using 90 flight schedule replications representing various demand levels.

Chapter 7 presents a summary of this research and conclueds with a discussuion on some possible research extentions.

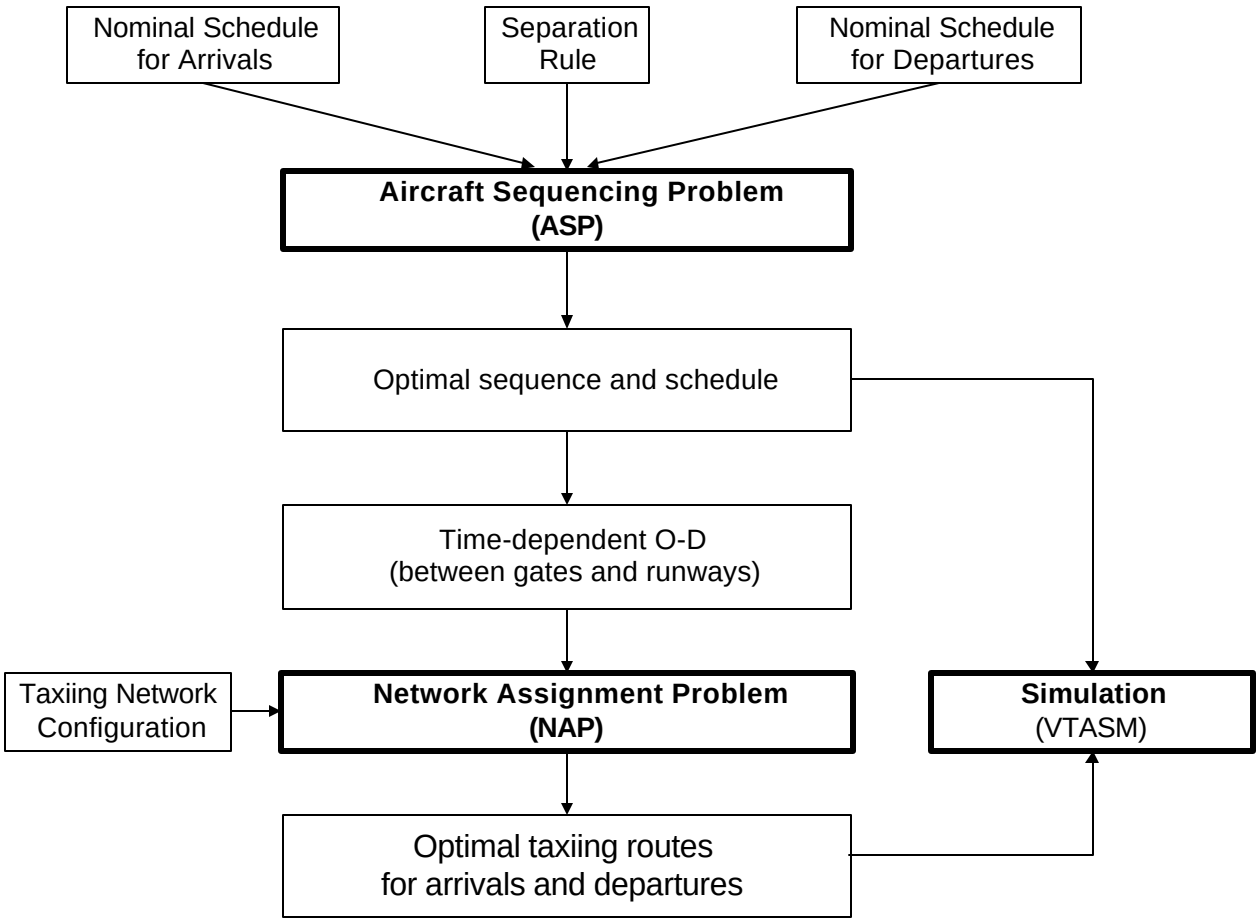


Figure 1.2 Problem Structure and Data Flow for the ASP and NAP Problem Framework (Single Data Horizon).

<Original schedule from flight-plan>

Flight No.	Aircraft type	Desired pushback Time
NW122	B747	12:37
NW123	B757	12:43
……..	….	….

<Original Schedule from data link>

Flight No.	Aircraft type	Desired touchdown Time
US322	B727	12:39
US321	B737	12:41
….	….	….

Ready-time to takeoff =
desired pushback time + taxiing time (T_1)

< Original Sequence>

max. delay =10 min.

Departures				Arrivals			
Flight no.	Aircraft type	Ready-time to takeoff	Due-time to takeoff	Flight No.	Aircraft type	Ready-time to touchdown	Due-time to touchdown
NW122	B747	12:47	12:57	US322	B727	12:39	12:49
NW123	B757	12:53	13:03	US321	B737	12:41	12:51
….	….	….	….	….	….	….	….

Min. Separation Rule → **ASP**

Minimize the completion time
for all aircraft to be operated

<Optimal Sequence>

Departure				Arrival			
Flight No.	Aircraft type	Opt. Time to takeoff	Delay	Flight No.	Aircraft type	Opt. time to touchdown	Delay
NW123	B757	12:53		US321	B737	12:41	
NW122	B747	12:55		US321	B727	12:46	
….	…….	….	….	….	….	….	….

Recommended push-back time from the gate =
Optimal takeoff time – Taxiing time – Buffer time

Expected time to exit from the runway =
Optimal touchdown time + Runway Occ. Time (ROT)

<Time-dependent taxiing demand>

Departures					Arrivals				
Flight No.	Aircraft type	Rec. time to pushback	Origin gate	Destination runway	Flight No.	Aircraft type	Exp. time to exit from runway	Origin runway exit	Destination gate
NW123	B757	12:49	Gate # 3	22R	US321	B737	12:41	Exit 2 on 22R	Gate # 9
NW122	B747	12:45	Gate # 5	22L	US321	B727	12:46	Exit 3 on 22L	Gate # 6
….	….	….	….	….	….	….	….	….	….

Assumption: Aircraft start taxiing at the recommended
pushback times/expected exit times

NAP

<Time-dependent taxiing demand>

Departures				Arrivals			
Flight No.	Origin gate	Routes (links)	Destination runway	Flight No.	Origin runway-exit	Routes (links)	Destination gate
NW123	Gate # 3	B12-> B23-> ….->	22R	US321	Exit 2 on 22R	E1->E3-> ….->	Gate # 9
NW122	Gate # 5	C1-> B4-> ….->	22L	US321	Exit 3 on 22L	F5->E3-> ….->	Gate # 6
….	….	….	….	….	….	….	….

Figure 1.3 Data Flow in the ASP/NAP Framework for Sample Problem (Single Data Horizon).

Chapter 2. Literature Reviews

2.1 Literature Review on Aircraft Sequencing Problems (ASP)

The Aircraft Sequencing Problem (ASP) is a combinatorial problem for which no efficient algorithm (i.e., with computational time bounded by a polynomial in the size of the problem) is known. When faced with such a computationally hard (formally classified as NP-hard–see Garey and Johnson, 1979) problems, two approaches are possible:

1) Branch-and-bound methods applied with efficient procedures to evaluate lower/upper bound at every nodes, which reduce the effort from that of exhaustive enumeration. These methods are certain to produce an optimal solution, but in the worst-case scenario, they may require an exponential number of calculations.
2) Heuristic methods which provide an approximate solution in fast (polynomial) time. These methods may not produce an optimal solution, but generate sub-optimal solutions that are close to an optimum for most cases.

In this review, we survey 1) the previous studies on Aircraft Sequencing Models (ASP), 2) a dynamic programming method along with branch-and-bound scheme, and 3) various types of heuristic methods for solving traveling salesman problems (TSP).

2.1.1 Previous Studies on ASP

The most straightforward sequencing strategy for arrivals is to assign the flights on a first-come-fist-serve (FCFS) basis, which is largely applied in air traffic control these days. In this method, the aircraft are scheduled in the order of preferred landing/departure times (or sometimes termed nominal landing/departure times). The second sequencing method is the so-called time advance (TA) technique which is used to minimize the average delay (or maximize throughput) without changing the order of the aircraft. In this method, the runway utilization is improved by speeding up aircraft during periods of heavy traffic so as to reduce the gaps that occur otherwise [Neuman and Erzberger, 1990]. The FCFS sequencing procedure is most effective in the sense of implementation, but it is subject to the randomness of the arrival process, and consequently, may create undesirable delays which can be reduced by an optimized sequence.

ASP problems are combinatorial problems where n aircraft have to be sequenced optimally. To enumerate all possible cases involves evaluating $n!$ solutions. Since the computations for a total enumeration increases drastically as n increases, we need to consider more constructive enumeration methods. Due to the nature of the problem, previous research efforts have focused on methods to reduce the computational complexity without losing the capability to obtain an optimal solution.

Attempts to optimize ASP-type problems date back to the late 60's. A first investigation of the ASP problem is made by Dear (1976). The main scope of his study is to determine the landing order of arriving aircraft considering all the aircraft currently in the system. As this number can be large (20 or even more simultaneous aircraft), Dear realizes the difficulty in attaining an optimal solution in real-time. To solve this problem, a Constrained Position Shifting (CPS) strategy is used instead of a FCFS strategy. That is, no aircraft may be sequenced forward or backward more than a pre-specified number of positions (Maximum Position Shifting) from its FCFS position. The method examines a window of ($2 \times$ the maximum position shift $- 1$) positions, optimizes it (exhaustively) for a single position shift, moves the window down one position, and repeats the process.

Considering Dear's CPS management concept, Psaraftis (1980) develops a dynamic programming approach for sequencing a given set of jobs on a single machine to minimize the total processing cost. In this paper, Psaraftis assumes that there are n distinct groups of jobs, where the jobs within each group are identical. Taking advantage of this grouping assumption, he suggests a more practical algorithm which can save some computational effort as compared to the classical dynamic programming approach for the job sequencing problem. He illustrates this method by optimally sequencing aircraft arrivals at an airport.

Using an integer programming approach, Bianco et al. (1987, 1997) determine an optimal sequence for arriving aircraft inside the Terminal Management Area (TMA). The authors point out the static nature of Psaraftis' study where all aircraft are supposed to wait to land at a given time. Based on this concept, this paper presents a formulation which takes into account the dynamic nature of the problem in which every aircraft entering the TMA has a Nominal Landing Time (NLT) depending on the characteristics of the TMA, the aircraft speed, and so on. The formulation developed by Bianco et al. (1987) is as follows.

Minimize $s + \sum_{i \in J_0} \sum_{j \in J_0} p_{ij} x_{ij}$

subject to

$$t_i + \sum_{j \in J_0} p_{ij} x_{ij} - \sum_{k \in J_0} \sum_{j \in J_0} p_{kj} x_{kj} - s \leq 0 \qquad i \in J_0 \qquad (1)$$

$$r_i - t_i \leq 0 \qquad i \in J_0 \qquad (2)$$

$$(p_{ij} + T_{ij}) x_{ij} + t_i - t_j - T_{ij} \leq 0 \qquad i \in J_0, j \in J, j \neq i \qquad (3)$$

$$\sum_{j \in J_0} x_{ij} = 1 \qquad i \in J_0 \qquad (4)$$

$$\sum_{i \in J_0} x_{ij} = 1 \qquad j \in J_0 \qquad (5)$$

$$s \geq 0 \qquad (6)$$

$$t_0 = 0 \qquad (7)$$

$$x_{ij} \in \{0,1\} \; i \in J_0, \; j \in J_0, \; j \neq i \qquad (8)$$

$$t_i \geq 0, \; i \in J_0$$

where,

$J = \{1,...,n\}$: a set of n jobs to be processed on a single runway

r_i : ready-time for job i

p_{ij} : processing time of job i if job j is the successor of i in sequence,

$i, j \in J_0$, $J_0 = J \cup \{0\}$

p_{0i} : setup time of the machine when the sequence starts with job i

x_{ij} = 1 if job i directly precedes job j, and otherwise 0

t_i = start time of job i

s = machine idle time

T_{ij} = sufficiently large value to make constraints (4) redundant whenever x_{ij}=0.

This problem is NP-Hard and in the case of zero ready-times it reduces to the Asymmetric Traveling Salesman Problem (ATSP). Here, Constraint (3) secures the minimum separation between two consecutive flights and also prevents subtours. The authors suggest a branching strategy based on the characteristics of the subsequences obtained in the solution process, and a Lagrangean lower bounding strategy. Heuristic upper bounds are also computed. Using their branch-and-bound approach, they find an exact optimal solution for the problems having up to 44 aircraft in 1,956 seconds, and show that an optimal solution could save up to 20 % on the runway utilization. Perhaps the major weaknesses in this formulation is the lack of due-time constraints for each flight, resulting in potentially unacceptable air delays, and the neglecting of non-consecutive separation restrictions.

One thing which should be noted here is that previous studies have mainly focused on the arrival operations. In Dear's study, an extension to mixed operations (i.e., departures and arrivals) is given, with one critical assumption. For simplicity of computation, Dear assumes a slightly relaxed separation in order not to create difficulties when the resulting sequence has two different types of consecutive operations, which may otherwise violate the separation rules. (More details of this issue will be discussed later.)

2.1.2 Dynamic Programming Approach

Sequencing and scheduling problems arise in a wide range of economic activities. The field of sequencing and scheduling theory has originated from the endeavor to solve mainly job-machine problems which involve accomplishing a number of tasks (jobs) that need to be processed by various resources (machines) according to certain specified rules over a period of time. If we view aircraft operations as jobs and a runway as a machine, we can apply job scheduling theory to our problem, which is to minimize the total delay time or completion time of all aircraft. In this problem, each aircraft has a ready-time, namely, the preferred time to land or depart.

Among the first analytical methods applied to solve sequencing problems was Dynamic Programming, which originated from the work of Richard Bellman in the fifties. Held and Karp (1962) applied Dynamic Programming to solve sequencing problems. Their approach is based on Bellman's "Principle of Optimality". Roughly, this says that in an optimal sequence, the first k aircraft (for any $k = 1,..., n$) must form an optimal sequence for the reduced problem based on these k aircraft alone. Dynamic Programming solves the problem in stages via a sequence of recursive computations in a manner that yields an optimal solution to the entire problem which is represented by the final stage.

To apply dynamic programming, we can divide our ASP into two sub-problems. First, for a single aircraft problem, it is necessary to consider simply the preferred landing/departing time (PLT/PDT) and the runway occupancy time. Second, for more than one aircraft, we need to consider a recursive equation. If we have n aircraft to be sequenced, $\{j_1, j_2, \ldots, j_k, \ldots j_n\}$, the equations for Dynamic Programming to minimize the completion time are as follows:

Let

J : the subset of n aircraft
$j(J)$: last aircraft in sequence of subset J
$j^*(J)$: optimal last aircraft in sequence of subset J
j^{**} : optimal second-to-last aircraft in subset J, given that aircraft j is last

(= $j^*(J-\{j\})$)

ROT_i : the runway occupancy time of aircraft i

p_{ij} : minimum separation time between aircraft i and j when aircraft j immediately follows i

$C(J, j)$: optimal completion time for subset J, given that the last aircraft is j

$C(J)$: optimal completion time for subset J

$f_j(C_j)$: cost such as delay time for aircraft j, given its completion time (C_j)

$G(J)$: optimal cost such as total delay time for subset J.

If J contains a single aircraft, i.e. $J = \{ j_k \}$

$$C(J, j) = G(j_k)$$
$$= PLT \text{ (or } PDT) + ROT_k,$$
$$G(J) = C(J, j_k)$$
$$= G(j_k),$$
$$j(J) = j_k,$$
$$j^*(J) = j_k.$$

else if J contains more than one aircraft (recursive equations)

$$C(J, j) = C(J-\{j\}) + p_{j^{**}j} + ROT_j,$$
$$G(J) = min_{\ j \text{ in } J} [G(J-\{j\}) + f_j(C(J, j))]$$
$$= [G(J-\{j^*\}) + f_{j^*}(C(J, j^*))],$$
$$C(J) = C(J, j^*(J)).$$

The branch-and-bound approach is a preferred solution technique over Dynamic Programming. As its name implies, the method consists of two fundamental features: 1) branching: this is the process of partitioning a large problem into two or more subproblems, and 2) bounding: this is the process of calculating a lower bound on the optimal solution value of a given subproblem.

As an example of a branching procedure, let P denote our ASP containing n jobs. The problem P can be solved by solving n related subproblems, P_1, P_2,…, P_n (subproblem P_j means job j has been constrained to come first, but all other aircraft are open for future assignments). The set of subproblems P_j is a mutually exclusive and exhaustive partition of P in the sense that if each P_j is solved, the best of these n solutions will represent an optimal solution to P.

Next, each of the subproblems can be partitioned, for instance, P_1 can be partitioned into P_{12}, P_{13}, … , P_{1n}. (In P_{12}, jobs 1 and 2 occupy the first two positions in the sequence). Therefore the second-level partition P_{1j} bears the same relation to P_1 as the first-level partition P_j bears to P. At level k, each subproblem contains k fixed positions and can be further partitioned into $(n-k)$ subproblems. If this branching were to be carried out completely, there would be $n!$ subproblems at level n. Clearly, evaluating all $n!$ cases is not practical, and so, a fathoming or curtailing of this enumeration tree needs to be devised.

Suppose that at some intermediate stage, a complete solution has been obtained that has an associated performance measure (called incumbent solution) Z. Also, suppose that a subproblem encountered in the branching process has an associated lower bound $lb > Z$. Then that subproblem need not be considered any further. To find a lower bound at a certain node A at level k (i.e., for the subproblem $P_{12\ldots k}$), we can use the following equation:

$$lb(A) = ROT_1 + P_{12} + \ldots + P_{(k-1)k} + (n-k)\cdot P_{min}^{(A)}$$

where,

ROT_1: the runway occupancy time of aircraft 1 if this is the first in the processing sequence

P_{ij} : the processing time (minimum separation) of aircraft j if it immediately follows aircraft i

$P_{min}^{(A)} = min_{\forall (i,j) \notin \{(1,2),\ldots,(k-1,k)\}}\{P_{ij}\}$.

Generally, the efficiency of the Branch-and-Bound method is strongly dependent on the trial solution and the tightness of the lower bound computed at each node. Unfortunately using $P_{min}^{(A)}$ makes it difficult to obtain a sharp lower bound at earlier nodes in the tree, implying that the method might require considerable computational time.

The Dynamic Programming method reduces the number of computations required to find an optimal sequence when compared to exhaustive enumeration. It is also true that the larger the problem, the more dramatic is the gain in speed. However, because Dynamic Programming must keep all the information at each and every stage and none of this information is redundant until the final identification of an optimal sequence, the storage requirements grow rapidly with the number of jobs. Considering the real-time requirements of our ASP problem, we need to devise more efficient solution methods. For this reason, this research suggests an Integer Programming approach as well as a heuristic approach to solve the aircraft scheduling and sequencing problem.

2.1.3 Reformulation-Linearization Technique (RLT)

RLT was developed by Sherali and Adams (1989, 1990, 1994), and is an automatic reformulation technique that can be used to derive tight LP representations as well as strong valid inequalities [Sherali and Adams, 1999]. Consider a mixed-integer zero-one programming problem whose feasible region X is defined as follow:

$$X = \{(x,y) \in R^n \times R^m : Ax + Dy \geq b, 0 \leq x \leq e_n, x \text{ binary}, y \geq 0\},$$

where e_n is a vector of ones in R^n.

Here, $x = (x_1,\ldots, x_n)$ is a set of binary variables and $y = (y_1,\ldots, y_m)$ is a set of continuous variables. RLT consists of essentially two steps: 1) a reformulation step in which additional non-linear valid inequalities are automatically generated, and 2) a linearization step in which each product term is replaced by a single continuous variable.

In the reformulation step, given $d \in \{1,...,n\}$, the RLT procedure constructs so-called "bound-factors (F_d) of degree d" comprised of the product of some d binary variables or their complements. These factors are then used to multiply each of the constraints defining X (including the variable bounding restrictions) to create a nonlinear polynomial mixed-integer zero-one programming problem. The bound-factors are defined as follow:

$$F_d(J_1,J_2) = \left[\prod_{j\in J_1} x_j\right]\left[\prod_{j\in J_2}(1-x_j)\right], \qquad \forall J_1,J_2 \subseteq N \equiv \{1,...,n\},\ J_1 \cap J_2 = \varnothing,\ |J_1 \cup J_2| = d\ .$$

In the linearization step, using the relationship $x_j^2 = x_j$ for each binary variable x_j, j = 1,…,n, substituting a variable w_J and v_{Jk}, respectively, in the place of each nonlinear term of the type $\prod_{j\in J} x_j\ \forall J \subseteq N$ and $y_k \prod_{j\in J} x_j\ \forall J \subseteq N, \forall k$, and relaxing integrality, the nonlinear polynomial problem is re-linearized into a higher dimensional polynomial set X_d defined in terms of the original variables (x, y) and the new variables (w, v). Sherali and Adams show that the projection of X_d onto the space of the original variables (x, y), X_{Pd}, yields the hierarchy of relaxations.

$$X_{P0} \equiv X_0 \supseteq X_{P1} \supseteq X_{P2} \supseteq \ldots\ldots \supseteq X_{Pn} = conv(X)$$

where, $X_{P0} \equiv X_0$ (for d=0) is the ordinary LP relaxation, and

$conv(X)$ is the convex hull of X.

The hierarchy of higher-dimensional representations produced in this manner markedly strengthen the usual relaxations, as is evidenced by the fact that the convex hull representation is obtained at the highest level. In fact, in computational studies on many classes of problems, even the first-level representation helps design algorithms that significantly dominate existing procedures.

Applications of RLT to specific problems include zero-one quadratic problems; zero-one quadratic assignment problems (Adams and Sherali, 1986); continuous and discrete bilinear

programming problems (Adams and Sherali, 1993, Sherali and Alameddine, 1992); continuous and discrete location-allocation problems (Sherali and Tuncbilek, 1995); polynomial programming problems (Sherali and Tuncbilek, 1995, 1997a, 1997b); factorable nonlinear problems (Sherali and Wang, 1999): 0-1 mixed integer problems with application to some specially structured problems (Sherali, et al., 1998): Miller-Tucker-Zemlin formulations for asymmetric traveling salesman problems (Sherali and Driscoll, 1999), among many others.

2.1.4 Heuristic Approaches for the Traveling Salesman Problem (TSP)

The ASP problem can be seen as a variation of Traveling Salesman Problem (TSP) for which no efficient algorithm (i.e., with computational time bounded by a polynomial in the size of the problem) is known. In this section, we present several heuristic methods to solve the TSP. Over past two decades, there have been several heuristic techniques developed that bear importance to our original problem.

2.1.4.1 Definitions of Various Types of Traveling Salesman Problems

Depending on the characteristics of the edge weight, w_{ij} (or t_{ij}), and time constraints on customer service times, traveling salesman problems are classified as Symmetric (henceforth abbreviated TSP), Asymmetric (ATSP), Symmetric with time-windows (TSP-TW), and Asymmetric with time-windows (ATSP-TW) traveling salesman problem. The definitions of various types of TSP problems are summarized in Table 2.1

2.1.4.2 Heuristic Approaches for the Traveling Salesman Problem (TSP)

Heuristic methods for TSP can be conveniently divided into two procedures: 1) *tour construction procedures* which *construct* a initial feasible solution, and 2) *Tour improvement procedures* which try to *improve* a given initial solution in a systematical way [Lawler, 1985].

2.1.4.2.1 Tour Construction Procedures for TSP

Reinelt (1994) classifies various types of construction heuristics suggested for TSP into four groups: nearest neighbor heuristics, insertion heuristics, heuristics using spanning trees, and

saving heuristics. Among the construction heuristics, the insertion method is known as one of the most efficient approximate algorithms for the initial tour construction [Syslo, 1983]. In general,

Table 2.1 Definitions of Various Types of TSP Problems.

Problem	Definition
TSP [1)]	*Given* a complete weighted digraph $G=(V, E)$ and symmetric weights w_{ij} (or t_{ij}) for all directed edges $(i, j) \in V \times V$ in E. *Find* a minimal Hamiltonian tour (or circuit) of G, i.e., a cycle which visits each vertex exactly once, having minimum total weight. (Most TSPs assume that a given vertex, say vertex 0, will serve as the first and last vertex of any route and that the weight matrix (w_{ij}) satisfies the triangle inequality.)
ATSP [2)]	Same as TSP except that $G=(V, E)$ is defined having asymmetric weights w_{ij} and w_{ji} for the directed edges (i, j) and (j, i) respectively. (i.e., w_{ij} is not necessarily equal to w_{ji}.)
TSP-TW [3)]	*Given* a complete weighted digraph $G=(V, E)$ with symmetric weights w_{ij} for all directed edges $(i, j) \in V \times V$ in E, and a time-window on the departure time for each vertex i denoted by $[e_i, l_i]$ where e_i specifies the earliest service time and l_i the latest service time. *Find* a minimal Hamiltonian tour of G satisfying the time-window restrictions.
ATSP-TW [2)]	Same as TSP-TW except that $G=(V, E)$ has asymmetric weights w_{ij} and w_{ji} for the directed edges (i, j) and (j, i) respectively.

1) Kanellakis (1980), Savelsberg (1990)
2) Acheuer (1999)
3) Savelsberg (1992)

an insertion method starts with a subtour comprising of one or two nodes, and extends the tour by selecting a node from currently unvisited nodes and then inserting the selected node to the current subtour in such a way that the incremental tour cost is as small as possible.

Let V be the entire node set where $|V| = N$ and V_T be the set of nodes included in the current subtour. The k^{th} iteration ($1 \le k \le N$-1) of an insertion algorithm enlarges the tour of size k to one of size k+1 by applying the following two steps:

Selection step: In the set $V\text{-}V_T$ of the unvisited nodes, determine which node is to be added to the cycle next.

Insertion step: Determine where the newly selected node is to be inserted to enlarge the current subtour.

For the selection step, a number of heuristics have been suggested and investigated. Some of these are:

arbitrary insertion: pick any unvisited node at random,

nearest insertion: pick the unvisited node that is nearest to the current subtour,

cheapest insertion: compare the cost of insertions of all unvisited nodes (in all insertion positions) and pick the one having the smallest cost,

farthest insertion: pick the unvisited node that is farthest from the current subtour.

In addition to the above methods, the convex hull insertion method, and the greatest insertion procedure can be applied, especially in an Euclidean space. (For more details, see Bodin (1983), Golden and Stewart (1985), and Laporte (1992a).)

Of the insertion heuristics mentioned above, the farthest insertion appears to be the best overall strategy (Golden et al., 1980). The underlying intuition behind the farthest insertion method is that if a rough outline of the tour can be constructed through the widely-spread nodes, then the finer details of the tour resulting from the incursion of the nearest nodes can be filled in without greatly increasing the total length of the tour.

The farthest insertion algorithms are described as follows [Syslo et al., 1983]: in order to find the farthest unvisited node, the array $dist(\cdot)$ of size N is maintained such that, for all unvisited nodes v, $dist(v)$ is the distance from the node v to the node in the current tour which is closest. The node f denotes the farthest node from the current tour. Each time a new node is inserted into the cycle, the *dist* array is updated. To find the best insertion position which is the closest edge from the node f, the insertion cost of node f between node i and j (IC_{ij}) is examined for all edges in the current subtour. The insertion cost is expressed as

$$IC_{ij} = w_{if} + w_{fj} - w_{ij}.$$

Once the closest edge is obtained, state variables such as the total cost, the array $dist(\cdot)$ and the node/edge sets corresponding to the current tour are updated. Below is the pseudo-code for the

farthest insertion algorithm. Here, s denotes an arbitrary node for an initial tour, and E_T is the edge set for the current subtour.

Initialization: $V_T = \{s\}$; $E_T = \{(s,s)\}$; $w_{ss} \leftarrow 0$; totalCost $\leftarrow 0$;
for all $u \in V\text{-}V_T$, $dist(u) \leftarrow w_{su}$;
Iteration:
while $|V_T| < n$
(*Selection*):
$f \leftarrow$ node in $V\text{-}V_T$ with largest value of $dist(f)$;
(*Insertion*):
for every edge $(i, j) \in E_T$, $IC_{ij} \leftarrow w_{if} + w_{fj} - w_{ij}$; // examine insertion costs
$(i^*, j^*) \leftarrow$ edge in E_T with smallest values of c_{ij}; // find the closest edge
(*update*):
$E_T \leftarrow E_T \cup \{(i^*, f), (f, j^*)\}\text{-}\{(i^*, j^*)\}$; // update the visited edge list
$V_T \leftarrow V_T \cup \{f\}$; // update the visited vertex list
$totalCost \leftarrow totalCost + c_{i^*j^*}$; // update the total tour cost
for all $x \in V\text{-}V_T$, $dist(x) \leftarrow \min\{dist(x), w_{fx}\}$ //update the array $dist$
end

2.1.4.2.2 Improvement Procedures for TSP

There is a wealth of previous studies on heuristic approaches to improve solutions to the traveling salesman problem. The best-known improvement heuristic procedures are edge-exchange (or sometimes called edge-interchange) methods. Lin (1965) proposed the r-opt algorithm in which r edges in a feasible tour are exchanged for other r edges in that tour as long as the result remains a feasible tour and the length of the exchanged tour is less than the length of the previous tour. Here, r is the number of edges exchanged at each iteration. The r-opt procedure is said to be r-optimal if there is no feasible exchange that improves the current solution. The larger the value of r, the more likely it is that the final solution is optimal. However, the computational requirements increase rapidly as the value of r increases. As a result, $r = 2$ or $r = 3$ are the ones most commonly used [Golden and Stewart (1985)].

A 2-exchange shown in Figure 2.1 involves the substitution of two edges, $(i, i+1)$ and $(j, j+1)$ with two other edges (i, j) and $(i+1, j+1)$. Such an exchange results in a local improvement if and only if

$$w_{i,j} + w_{i+1,j+1} < w_{i,i+1} + w_{j,j+1} .$$

Testing this improvement involves only local information and can be done in a constant time. The total number of possible 2-exchanges is equal to ${}^{N}C_2$, which implies a time complexity of $O(N^2)$ for the verification of 2-optimality.

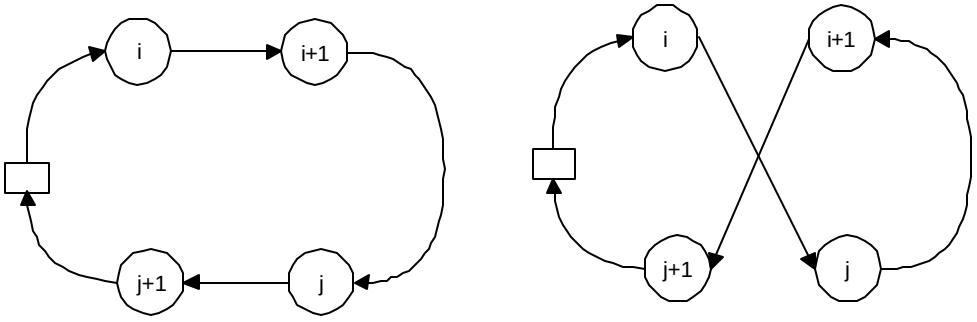
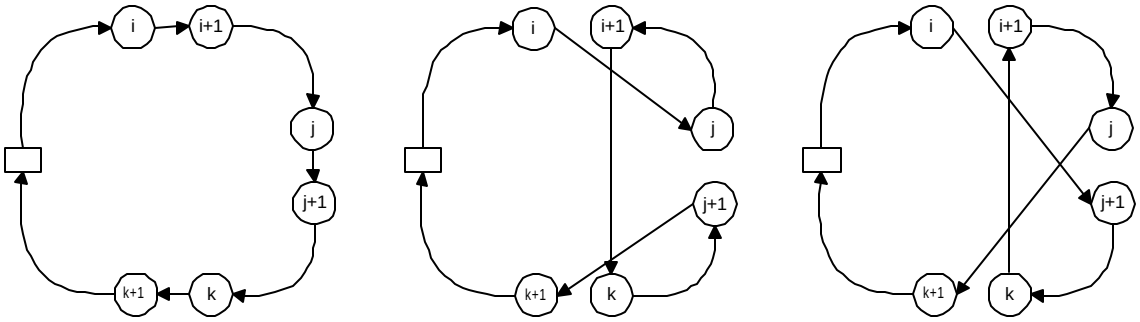


Figure 2.1 Example of 2-exchange.

It should be noted that the orientation of the path (i+1,…, j) is reversed in the proposed 2-exchange tour. In a TSP having side-constraints such as time-windows, it is of importance to be able to efficiently check the feasibility of the nodes in the reversed path. (This will be discussed later.)

In a 3-exchange, where three edges are removed, there are several ways to construct a new route. Two 3-exchanges shown in Figures 2.2(b) and (c) make an important difference: In Figure 2.2(c) the orientation of the original tour is preserved whereas in Figure 2.2(b) this orientation is reversed. The time to verification of 3-optimality is $O(N^3)$.



(a) (b) (c)

Figure 2.2 Example of 3-exchange.

Lin and Kernighan (1973) propose a variable *r*-opt algorithm which decides at each iteration how many edges to exchange. The variable *r*-opt requires considerably more effort in coding than the standard *r*-opt procedure. But the variable *r*-opt procedure outperforms the standard 2-opt or 3-opt approach in finding near-optimal solutions. Because of this advantage, variable *r*-opt procedure is frequently used to produce tighter upper bounds for the TSP.

With an intention to reduce the computational burden to verify *r*-optimality, proposals have been made to take only a subset of all possible *r*-exchanges into account. Or (1976) proposes a method that considers only those *r*-exchanges that would result in a string of *s* (= 2, 3..) consecutive nodes being inserted between two other nodes. This reduces the time complexity required for the verification of Or-optimality to $O(N^2)$. Figure 2.3 shows an example of Or-exchange (*s*=2) where a string of (i_1, i_2) is removed from its position and inserted between *j* and *j*+1.

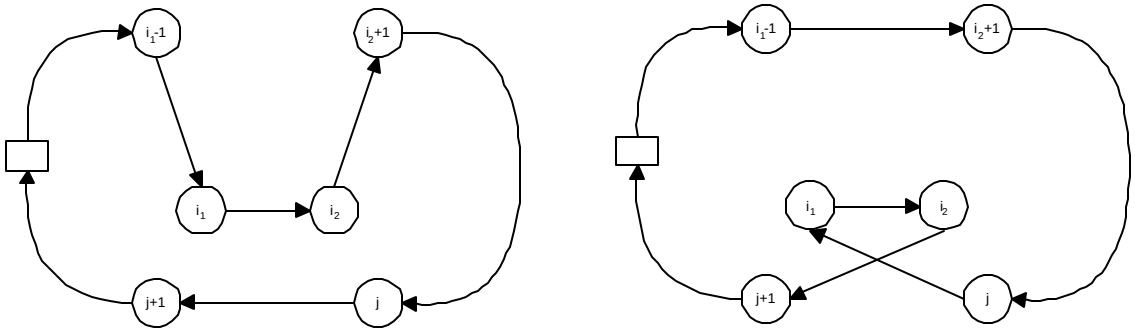


Figure 2.3 Example of an Or-exchange (s = 2).

2.1.4.3 Heuristic Approaches for the Traveling Salesman Problem with Time-windows (TSP-TW)

Due to time-window restrictions at each node (or customer location), an arrival earlier than the earliest service time e_i introduces a waiting time at node *i*. On the other hand, the tour is infeasible if the arrival time at node *i* is later than the latest service time l_i. Taking the time-window into consideration, there are two types of objectives for TSP-TW [Savelsbergh, 1985]:

Minimizing the total travel time, $\sum_{k=0}^{n} t_{k,k+1}$, which does not consider any possible waiting times at the nodes.

Minimizing the route duration, A_{n+1}-D_0, i.e., the time difference between the arrival time at the depot (A_{n+1}) and the departure time at the depot (D_0).

If the departure time at the initial node (node zero) is assumed to be set at its earliest time (i.e., $D_0=e_0$), the objective of minimizing the route duration becomes that of minimizing the completion time of the tour. This is suitable for our ASP problem which minimizes the total completion time of runway operations for a given set of flights. In this review, we focus on the procedures for the objective of minimizing the total route duration.

Most procedures for TSP can be successfully extended to TSP-TW. However, one difficulty in applying heuristics designed for TSP to TSP-TW is testing the feasibility of a candidate tour produced by any exchange scheme. For example, testing the feasibility of a k-exchange tour in TSP with time-windows (or other side-constraints) requires $O(N)$ time which results in a time complexity of $O(N^{k+1})$ for the verification of k-optimality. Various types of researchers have focused on devising more efficient techniques to test this feasibility.

By adopting the GENIUS (Generalized Insertion and Unstring and String) method for TSPs, Gendreu et al. (1998) proposed a generalized insertion heuristics for TSP-TW in which the objective is the minimization of travel time. The tour building algorithm produces a feasible route by inserting a vertex in its neighborhood on the current route, and performing a local optimization. Then, the feasible tour is post-optimized based on the successive removal and reinsertion of all vertices.

Since the early eighties, TSP-TW has drawn additional attention due to the fact that procedures for TSP-TW provide basic methods to solve *vehicle routing problem with time-windows* (VRP-TW). These are very important and practical problems faced by the industries. The *vehicle routing problem* (VRP) is a problem to design a set of minimum cost vehicle routes for a fleet of vehicles of known capacity which service a set of customers with known demands [Solomon et al., 1988]. All routes must originate and terminate at a common depot. Each customer is served exactly once. In addition, all N customers must be assigned to vehicles such that the total demand on any route does not exceed the capacity of the vehicle assigned to that route. The *vehicle routing problem with time-windows* (VRP-TW) is a generalization of the VRP. In the VRP-TW, a number of customers have one or more time-windows during which service must be scheduled. Most VRP-TWs assume that the number of vehicles used is unlimited, i.e., the fleet size is determined simultaneously.

Similar to TSP-TW, of primary importance to the effectiveness and efficiency of heuristics for VRP-TW is the way in which the time-window constraints are incorporated in the solution

process [Solomon, 1987]. (In VRP, it is true that we should consider the *vehicle capacity* constraint as well as *time-windows*. But, here we will skip discussions about the capacity constraints. For more details about capacity constraint, see Savelsbergh, 1990a).

If we have a single vehicle to be scheduled (i.e. single VRP-TW), the problem then becomes the same as TSP-TW. For this reason, it is worth reviewing previous studies on VRP-TW. These studies include Psaraftis (1983a, 1983b), Savelsbergh (1985, 1990a, 1990b, 1992), Solomon (1986, 1987) Solomon et al. (1988), Sexton and Bodin (1985), Baker and Schaffer (1986), Desrosiers et al. (1986), Dumas et al. (1991, 1995), Desrochers et al. (1992), and Laporte (1992b). Several of these procedures are reviewed below.

2.1.4.3.1 Psaraftis 's Procedure

A. Tour Building Procedures

Psaraftis (1983a) suggests an $O(N^2)$ heuristic for the dial-a-ride problem (DARP) which is a special type of TSP with precedence constraints. In the DARP, a vehicle is located at a point A, and is called to service N customers, each of whom wishes to travel from a distinct origin to distinct destination, and then returns to A so that total length of the route is minimized. Here, the precedence among nodes should be considered because no destination can be visited before the corresponding customer has been picked up. The problem is *static* in the sense that all N customers' requests are given and no new customer requests are considered until all of these N customers are serviced.

Due to the complexity of the DARP, it is extremely hard to find an exact solution in reasonable time for practical size problems (usually having more than 100 customers for a vehicle). For instance, using Dynamic Programming to solve the DARP requires $O(N^2 3^N)$ time, and this approach limits the tractable problem size to no more than 8-10 customers [Psaraftis, 1980].

The tour building procedure proposed by Psaraftis is based on the minimum spanning tree (MST) approach that is defined for N origins and N destinations. First, an initial TSP tour T_0 through the $2N$ nodes is constructed. Then subsequent steps produce a feasible tour by traversing T_0 in such a way that precedence constraints are satisfied. The heuristic proceeds as follows.

Step 1 (*Generate a TSP tour, T_0*): Without distinguishing origins and destinations, construct a TSP tour T_0 through $2N$ points based on their MST.

Step 2 (*Generate a dial-a-ride tour, T_1*): Choose any customer's origin on T_0 as a first pick-up point P_1 around the starting point A. Construct a dial-a-ride tour T_1, by traversing T_0 clockwise from P_1 until all points are visited and then return to A. While doing this, do not visit any point that has been previously visited or any destination whose origin has not been visited yet.

Step 3 (*Improve T_1: optional*): Improve T_1 by a sequence of a local exchanges.

Step 4 (*Generate a dial-a-ride tour T_2: optional*): Generate another tour by repeating Step 2 and Step 3 but moving counterclockwise. Pick the shortest tour in T_1 and T_2.

Step 5 (*Optional*): Repeat Step 2 (optionally 3 and 4) N times, each time choosing a different customer origin as P_1. Pick the tour that has the minimum length.

For Step 3, Psaraftis proposes a local improvement scheme in the sense that the exchange involves four adjacent nodes such as $i, j, k,$ and m shown in Figure 2.4. Assuming that the edge weights, w_{ij}, are symmetric and that the triangle inequality holds, the conditions for feasibility and profitability of the exchange shown in Figure 2.4 are as follow:

$w_{ij} + w_{km} < w_{ik} + w_{jm}$ (condition for profitability),

where, k is not the destination of the customer whose origin is j (condition for feasibility).

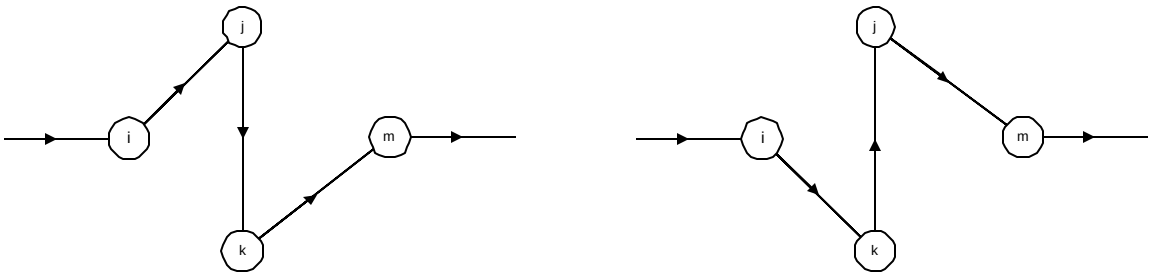


Figure 2.4 A local exchange [Psaraftis, 1983a].

The computational complexity of the heuristic is $O(N^2)$ since finding MST, along with Step 3 and 5 each require $O(N^2)$ time. (Even though the MST in Euclidean plane can be found in $O(NlogN)$ time, still the heuristic is $O(N^2)$.)

B. Improvement Procedures

In the context of DARP, Psaraftis (1983b) develops k-exchange procedures to perform local search in a precedence-constrained vehicle routing problem. Similar to the k-opt procedure of Lin (1965), and Lin and Kernighan (1973), a DARP tour is said to be k-optimal if it is impossible to obtain another DARP tour of shorter length by replacing any k of its links by any other set of k links.

Figure 2.5 depicts a 2-exchange. Such an exchange is profitable in a local tour improvement if and only if $w_{i,i+1}+w_{j,j+1} > w_{i,j}+w_{i+1,i+1}$ under the assumption that the triangle inequality holds. In contrast with the TSP where each individual exchange takes $O(1)$ time, checking whether each DARP exchange satisfies the origin-destination precedence constraints requires $O(N^2)$ time. (This is so, because checking for precedences needs an examination of all pairs of nodes in the section $(i+1, j)$ and to ascertain if there is any customer who has the corresponding origin and destination in the segment.) If the feasibility check is executed at every 2-inerchange, the procedure to find a 2-opt DRAP tour from a feasible DRAP tour will take $O(N^4)$ time.

Psaraftis proposes a method which finds the best 2-iterchange DARP tour out of a given feasible tour in $O(N^2)$ time, which is the same as in the TSP where no feasibility checks are needed. Suppose that we have an initial feasible DARP tour having N customers (labeled $n=1,\ldots,N$). A DARP tour can be represented in one of following two ways.

Either using an array for a sequence counter $S(i)$, $i=0,\ldots,2N+1$, representing the i^{th} stop of the tour given by following definition:

$$S(i) = \begin{cases} 0 & \text{if } i = 0 \text{ and } 2N+1 \text{ (i.e., the starting and ending point of the tour),} \\ +n & \text{if the vehicle picks up customer } n \text{ at stop } i, \\ -n & \text{if the vehicle delivers customer } n \text{ at stop } i. \end{cases}$$

Or using a matrix $[m(n,i)]$ in which $m(n,i)$ represents the status of customer n at the end of the i^{th} stop of the tour:

$$m(n,i) = \begin{cases} 3 & \text{if customer } n \text{ has not been picked up so far,} \\ 2 & \text{if customer } n \text{ is on board the vehicle,} \\ 1 & \text{if customer } n \text{ has been delivered.} \end{cases}$$

Let us consider a 2-exchange of $(i, i+1)$ and $(j, j+1)$ as shown in Figure 2.5. Checking the precedence feasibility of the 2-exchange can be done by simply examining if there is any customer n for whom $m(n, i+1) = 3$ and $m(n, j) = 1$. If there exists such a customer, the proposed

2-exchange will violate the precedence constraint. The matrix $[m(n, i)]$ can be constructed from array $S(i)$ in $O(N^2)$ time. Having this matrix, we can check the precedence feasibility of a proposed 2-exchange in $O(N)$ time. Hence, the best 2-exchange DARP tour from a given initial tour can be obtained in $O(N^3)$ time.

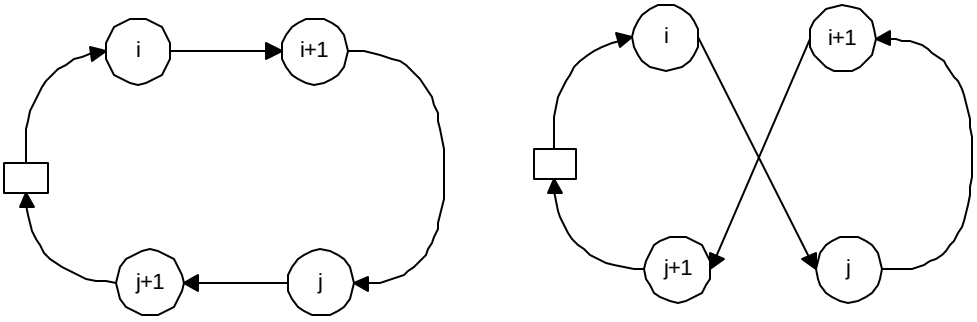


Figure 2.5 An example of 2-exchange.

In order to reduce the computational complexity further, Psaraftis introduces a screening procedure to determine the feasibility of every possible 2-intercange. This screening process is performed at the beginning of the algorithm, and the information is stored in a matrix to be used in the tour improvement procedure.

Given a DARP tour and a stop i ($0 \le i \le 2N$-2), let FIRSTDEL(i) denote the position (or stop) of the first delivery remaining beyond i+1, for which the corresponding customer has not been picked up until stop i. The FIRSTDEL(i) is expressed as follows:

$$\text{FIRSTDEL}(i) = \begin{cases} x & \text{if } x \text{ is the smallest position above } i+1 \text{ for which there exists a} \\ & \text{customer } n \text{ so that } m(n, i) = 3 \text{ and } m(n, x) = 1, \\ 2N+1, & \text{otherwise.} \end{cases}$$

Then, the precedence feasibility of a 2-exchange can be checked by applying the following theorem.

Theorem (Psaraftis (1983b)): The substitution of links $(i, i+1)$ and $(j, j+1)$ with (i, j) and $(i+1, j+1)$ is feasible if and only if $j <$ FIRSTDEL(i).

Using the array of FIRSTDEL(·) and the Theorem, the feasibilities of all possible 2-exchanges are stored in the matrix [FE(i, j)] with values of true or false. The pseudo-code for the screening process is presented below:

Step 1(*Calculate the values of* FIRSTDEL(i)): Using array $S(i)$ and the matrix [$m(n, i)$], calculate the values of FIRSTDEL(i) for all i, $0 \le i \le 2N$-2.

```
Do i = 0 to 2N-2
   Do x = i+2 to 2N+1
        If (S(x) < 0)            // check if x is delivery
           If m(-S(x), i) = 3) // check if x has already been picked up util the end of stop i
               FIRSTDEL(i) = x;
           End if
        End if
        If (x = 2N+1) then FIRSTDEL(i) = x;
    End do i
End do x
```

Step 2 (*Create a feasibility matrix* [FE(i, j)] *for all possible* 2*-exchanges*):

```
Do i = 0 to 2N-2
   Do j = i+2 to 2N
        FE(i, j) = false;
        If j < FIRSTDEL(i), then FE(i, j) = true;
   End do i
End do j
```

The above screening process can be executed in $O(N^2)$ time. As a result, the best 2-exchange tour out of a given DARP tour can be found in $O(N^2)$ time since checking feasibility of any proposed 2-exchange can be performed in $O(1)$ time. It should be emphasized that what we have found so far is not the find 2-opt tour but the best 2-interchange tour of a given DARP tour. To search for a 2-opt tour, the procedure should be applied a number of times. To facilitate this process, Psaraftis introduces two search algorithms: Breath-first and depth-first search (see Psaraftis (1983b) for a complete discussion of this).

2.1.4.3.2 Solomon's Procedures

A. Tour Building Procedures

By extending the known VRP heuristics, Solomon (1987) proposes several tour-building algorithms for VRP-TW. The novelty of the proposed approach is the incorporation of distance and time dimensions in the heuristic process. The cost (for the objective function) of direct travel from customer i to j is assumed to be given by $c_{ij} = r_1 w_{ij} + r_2 (b_j - b_i)$, where $r_1 \geq 0$, $r_2 \geq 0$ and w_{ij} is the direct travel time between i and j, and b_j is the time to begin service for customer j. If $r_1 = 0$, then the problem is to minimize the total travel time.

Solomon proposes necessary and sufficient conditions for time feasibility when inserting a customer, u, between the customers i_{p-1} and i_p, $1 \leq p \leq m$, on a partially constructed feasible route, $(i_0, i_1, i_2, \ldots, i_m)$, $i_0 = i_m = 0$ for which the times to begin service, b_{i_r} for $1 \leq r \leq m$, are known. Initially, the vehicle is assumed to leave the depot at the earliest possible time, e_0. (Later, the depot departure time is adjusted to eliminate any unnecessary waiting time after the complete vehicle schedule has been created.) The necessary and sufficient condition for feasibility of the insertion is as follows:

$b_u \leq l_u$ and $b_{i_r} + \mathrm{PF}_{i_r} \leq l_{i_r}$, $p \leq r \leq m$,

where, PF_{i_r} : the *push-forward* for customer i_r is defined as $b_{i_p}^{new} - b_{i_p} \geq 0$, if $r = p$, and $\max\{0, \mathrm{PF}_{i_r} - W_{i_{r+1}}\}$, if $p \leq r \leq m - 1$,

and where, $W_{i_{r+1}}$ is the waiting time at i_{r+1}.

It should be noticed that if $\mathrm{PF}_{i_p} > 0$, some of the customers i_r, $p \leq r \leq m$, could become infeasible. Hence, we need to examine these customers sequentially for time feasibility until we find a customer, i_r, for which $\mathrm{PF}_{i_0} = 0$, or i_r is time infeasible.

It should also be noticed that, the aforementioned condition assumes that the triangle inequality holds for travel distance and times. If non-Euclidean travel distances and times are used, then it is possible that $\mathrm{PF}_{i_p} < 0$, which leaves all the customers time feasible. Solomon proposes several types of heuristic methods for the tour building methods.

Saving heuristics:

This approach is an extension of the savings heuristic originally proposed by Clarke and Wright (1964). The procedure begins with n distinct routes in which each customer is served by a dedicated vehicle. The tour-building heuristic is performed by the addition of a link of distinct partially formed routes between two end customers, i and j, guided by a measure of cost savings given by $S_{ij} = d_{i0} + d_{0j} - m d_{ij}$, $m \geq 0$. In the VRP-TW, the route orientation must be considered

when two partial routes (see Figure 2.6) with end customers, *l*(*ast*) and *f*(*irst*), respectively, are combined according to the savings value. Here, testing for time feasibility can be accelerated by using the *push-forward* generated at *f*.

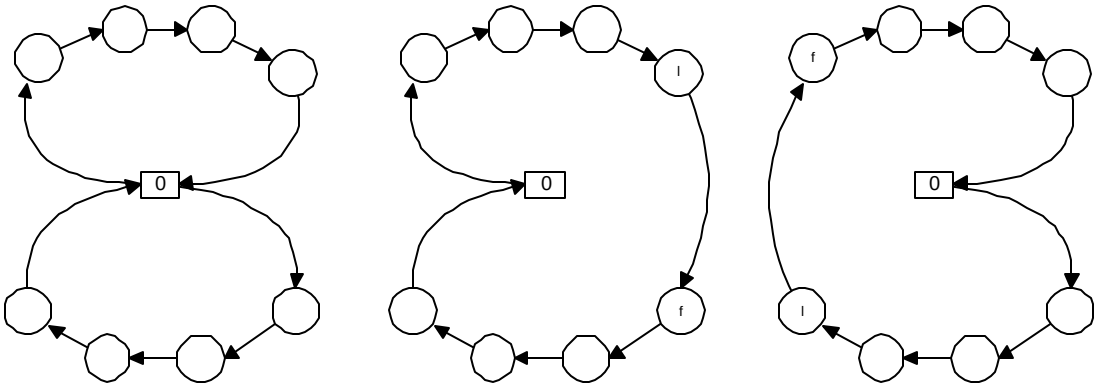


Figure 2.6 Two-addition procedures in a savings heuristic.

A time-oriented, nearest-neighbor heuristic:

This procedure initializes every route by finding the unrouted customer closest to the depot. The heuristic searches for the customer closest to the last customer added to the route. Let the last customer on the current partial route be customer i, and let j denote any unrouted customer. The closeness of any two customers, i and j, denoted by c_{ij}, is the combination of the distance between two customers d_{ij}, the time difference between the completion of service at i and the beginning of service at j, T_{ij}, and the urgency of delivery to customer j, v_{ij}, given by the following equations:

$T_{ij} = A_j - (A_i + s_i)$,

$v_{ij} = l_j - (A_i + s_i + w_{ij})$ and

$c_{ij} = d_1 w_{ij} + d_2 T_{ij} + d_3 v_{ij}$, where $d_1 + d_2 + d_3 = 1$, $d_1, d_2, d_3 \geq 0$,

where, A_i: arrival time at customer i,

s_j: service time for customer i,

B. Improvement Procedures

Suppose that we have a 2-exchage tour as shown in Figure 2.5. Similar to Psarsftis's procedure, Solomon (1988) considers two conditions to be satisfied for a substitution of (i, j) and $(i+1, j+1)$ with $(i, i+1)$ and $(j, j+1)$ to be favorable:

local improvement condition: $w_{ij}+w_{i+1,j+1} < w_{i,j+1} + w_{i+1,j}$,

feasibility condition: The time-window constraints of the customers affected by the exchange need to be satisfied.

To check the feasibility of the 2-exchange, time-windows for all customers from $i+1$ to the end of the route should be examined. Since this additional checking procedure requires $O(N)$ time, the total computational effort for 2-opt would result in a $O(N^3)$ time process.

By adopting the work of Psaraftis (1983b), Solomon et al. (1988) develop an acceleration method for improvement heuristics for VRP-TW. This method is used as a preprocessor, which makes it possible to handle the time-window constraints without an increase in running time of the algorithm. Checking the feasibility of a 2-exchnage can be accelerated by examining the precedence relationship between all pairs of customers. If $A_i+s_i+w_{ij}$ $(=D_i+w_{ij}) > l_j$, then customer i should precede customer j in the tour. The precedence information between all pairs of customers are stored in a matrix VP(i, j) in the following way:

$$\mathrm{VP}(i,j) = \begin{cases} +1 \text{ if customer } i \text{ must precede customer } j, \\ 0 \text{ if no precedence relationship exists,} \\ -1 \text{ if customer } j \text{ must precede customer } i. \end{cases}$$

Similar to Psaraftis' method, it is possible to define a node precedence value, NP(i), for all customers on the route which reflects the precedence dependence at that point with regard to customers to be visited later in the route.

NP(i) = the smallest number k, $k > i+1$, such that VP$(j, k) = +1$, $j \geq i+1$. If no such k exists, then NP$(i) = N+1$,
where, N is the number of customers.

Given NP(i) for all i, a necessary condition for the time feasibility of a 2-exchange is stated as follows:

A necessary condition for the feasibility of the 2-exchange of arcs $(i, i+1)$ and $(j, j+1)$ with (i, j) and $(i+1$ and $j+1)$ is that $j <$ NP(i).

It is noted that VP(i, j) may be examined in $O(N^2)$ time, and NP(i) may be obtained from the VP(i,j) in $O(N^2)$ time. It should also be noted that, unlike Psaraftis's procedure (1983b) for dial-a ride problem, the aforementioned condition is not a sufficient condition but only a necessary condition for feasibility of time-windows. Although the use of the NP(i) array does not eliminate the need for further checking of the feasibility of a 2-exchage, Solomon et al. suggest that it may be used as an effective filter to reduce the number of complete feasibility checks to obtain 2-optimal solution for the VRP-TW. (For 3-exchanges and some examples, see Solomon et al. (1988).)

2.1.4.3.3 Savelsbergh's Procedure

A. Tour Building Procedure

Savelsbergh (1990) introduce the *forward-time slack at node i*, F_i, to indicate how far the departure time of the node can be shifted in time without causing the route to become infeasible along the current path. The goal of the procedure is to check feasibility of the insertion of an unrouted customer u between two routed customers i and i+1. Taking the departure time at the initial node as the earliest service time (i.e., $D_0 = e_0$), F_i is defined as follows:

$$F_i = \min_{i \le k \le n} \left\{ l_k - \left(D_i + \sum_{p=i}^{k} w_{p,p+1} \right) \right\}.$$

Then, the feasibility of the insertion u will be secured if

$$\max\{D_i + w_{i,u}, e_u\} + w_{u,i+1} - D_{i+1} \le F_i.$$

Using the following backward recursion, F_k for all customers k can be computed in $O(n)$.

$$F_{n+1} = l_{n+1} - D_i ,$$

$$F_k = \min\{ F_{k+1} + W_{k+1},\ l_k - D_k \} \text{ for } k = n,\ldots 1,$$

where, W_i: waiting time at node i defined as $\max\{0, e_i - A_i\}$.

For the *profitability* of the insertion, Savelsbergh uses the measure of savings given by the following equation:

$$sav = 2w_{0,u} + w_{i,i+1} - w_{i,u} - w_{u,i+1}.$$

B. Tour Improvement Procedures

The basic idea of Savelsbergh's tour improvement procedure (1990b) is to use a specific search strategy in combination with a set of global variables such that testing the feasibility of a single exchange and maintaining the set of global variables requires no more than a constant time, $O(1)$. The set of global variables is defined such that: 1) this set of variables makes it possible to test the feasibility of a proposed exchange in constant time, and 2) the lexicographic search strategy makes it possible to update the values of these variables in constant time. Savelsbergh suggests three types of lexicographical search strategies for VRP-TW as shown in Figure 2.7.

Lexicographic search for 2-change: In the outer loop, choose the edge $(i, i+1)$ in the order in which they appear in the current tour starting at (0,1). For the inner loop, choose the edge $(j, j+1)$ to be $(i+2, i+3)$, $(i+3, i+4)$, …, $(n-1, n)$ (see Figure 2.7(a)).

Lexicographic search for backward Or-exchange: Choose the path $(i_1,\ldots,i_2)$ in the order of the current route starting with i_1 equal to 2. Choose the edge $(j, j+1)$ to be (i_1-2, i_1-1), (i_1-3, i_1-2), …, (0, 1) (see Figure 2.7(b)).

Lexicographic search for forward Or-exchange: Choose the path $(i_1,\ldots,i_2)$ in the order of the current route starting with i_1 equal to 1. Choose the edge $(j, j+1)$ to be (i_2+1, i_2+2), (i_2+2, i_2+3), …, $(n-1, n)$ (see Figure 2.7(c)).

A general framework for the 2-exchange procedure is roughly described by the following pseudo-code:

```
{ input: a route given as (0,1,2,…,n)}
{ output: a route that is 2-optimal}
Start:
for i=0 to n{
        initGlobal(i,G);
        for j=i+2,n{
                if(w_{i,j} + w_{i+1,j+1} < w_{i,i+1} + w_{j,j+1}  and FeasibleExchange(i,j,G))
                {
                    PerformExchange(i,j);
```

```
                Goto Start;
            }
            UpdateGlobal(i,j,G);
        }// end for j
    }// end for i
```

Comparing this procedure with the straightforward implementation, the suggested idea guarantees that only constant time is spent on a single exchange, which implies an $O(n^2)$ method for verifying 2-optimality. On the other hand, in the straightforward implementation, the time spent on a single exchange depends on the effort needed to establish either its feasibility or its infeasibility, which implies an $O(n^3)$ method for verifying 2-optimality.

Generally, the *forward-time slack* at node i related to the path $(i,\ldots, j)$ and to the departure times $D_i,\ldots, D_j$ is expressed as:

$$F_i^{(i,\ldots,j)} = \min_{i \le k \le j} \left\{ l_k - \left(D_i + \sum_{p=i}^{k} t_{p,p+1} \right) \right\}.$$

Savelsbergh (1992) proves that if two feasible path $(i_1,\ldots, j_1)$ and $(i_2,\ldots, j_2)$, with associated forward time slacks $F_{i_1}^{(i_1,\ldots,j_1)}$ and $F_{i_2}^{(i_2,\ldots,j_2)}$ for the initial nodes are concatenated, the *forward-time slack* for the first node of the resulting path is given by:

$$F_{i_1}^{(i_1,\ldots,\, j_1, i_2,\ldots,\, j_2)} = \min\left\{ F_{i_1}^{(i_1,\ldots,\, j_1)}, F_{i_1}^{(i_2,\ldots,\, j_2)} + \sum\nolimits_{i_1 < k \le j_1} W_k + D_{i_2} - (D_{j_1} + w_{j_1 j_2}) \right\}.$$

Using the above equation, we can compute the *forward-time slack* at the depot, $F_0^{(0,\ldots,n)}$, using one of the following two ways:

forward recursion: $F_0^{(0,\ldots,i,i+1)} = \min\{ F_0^{(0,\ldots,i)}, l_{i+1} - D_{i+1} + \sum_{0 < p \le i} W_p + W_{i+1} \}$.

backward recursion: $F_0^{(i,i+1,\ldots,n)} = \min\{ l_i - D_i, F_{i+1}^{(i+1,\ldots,n)} + W_{i+1} \}$.

Another issue in improvement procedures is how efficiently the objective function associated with a given exchange can be evaluated. Since our objective is to minimize the route duration, D_{n+1}-D_0, the efficiency of evaluation actually depends on the efficiency of the computation of

D_{n+1}. Given a path (i,…, j), a departure time D_i and D_0=e_0, the departure time D_j can be computed as

$$D_j = D_i + \sum_{i \le k < j} w_{k,k+1} + \sum_{i < k \le j} W_k \ .$$

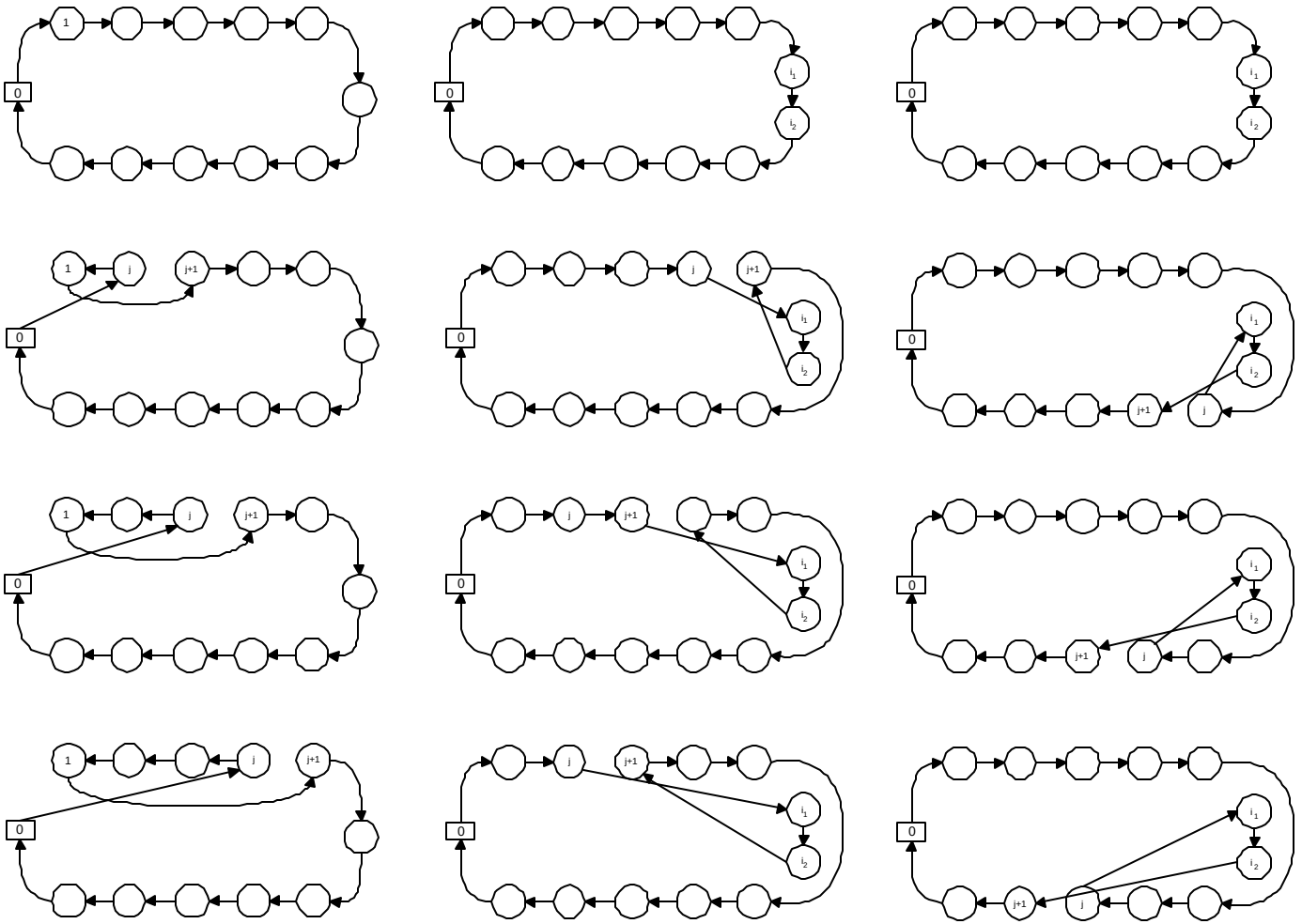
As such, the computation of waiting times, W_k, in a constant time is the real issue in checking for profitability. Savelsbergh (1992) suggests an approach to compute the waiting time on the concatenation of paths (i_1,…, j_1) and (i_2,…, j_2) by distinguishing four different cases on D which denotes $D_{j_1} + w_{j_1, j_2} - D_{i_2}$, and W_2 which is the sum of the waiting time on the path (i_2,…, j_2). Table 2.2 summarizes the results. Here B_2, which is the *backward-time slack* at node i relative to the path (i,…, j) indicates how far the departure time of the node can be shifted backward in time without introducing any waiting time, and is computed as follows:

$$B_i^{(i,\ldots,j)} = \min \sum_{i \le k \le j} \{D_k - e_k\}.$$

Table 2.2 Computation of Waiting Time on a Concatenated Path.

	$D \ge 0$	$D < 0$
$W_2 = 0$	W_1	$W_1+\max\{0, -D-B_2\}$
$W_2 > 0$	$W_1+\max\{0, W_2-D\}$	$W_1+ W_1 + D$

Savelsbergh shows that on the concatenated path, both the *forward-time slack* at each node that is used for the checking feasibility, and the sum of the waiting times used for checking the profitability can be computed in constant time. Hence, it is possible to implement the testing of feasibility and profitability in $O(1)$ time.



(a) 2-changes (b) Forward Or-exchanges (c) Backward Or-exchange

Figure 2.7 Lexicographic search strategies.

2.1.4.4 Heuristic Approaches to the Asymmetric Traveling Salesman Problems (ATSP)

In principle, if an asymmetric TSP is translated into a symmetric TSP, we can now apply any heuristic devised for symmetric TSP to solve ATSP. As pointed out by Kanellakis and Papadimitriou (1980), this approach increases the problem size considerably. For example, a 100-city ATSP is transformed into a 300-city symmetric TSP. They also remarked that not all approaches to the symmetric TSP can be adopted to solve the ATSP (for example, the class of techniques based on spanning trees). The power of Lin-Kernighan's (1973) heuristic is the fact that all primary changes are potentially searched. Kanellakis and Papadimitriou (1980) present an extension of Lin-Kernighan's local search algorithm for the solution of ATSP: the *sequential primary change* in which the creation of a new cycle (by primary change as described below) is immediately followed by a breaking of the cycle. They also suggest that the so-called *quad change* which is nonprimary can substantially enrich the neighborhood structure. A *quad change* is illustrated in Figure 2.8(d).

It is pointed out that Lin-Kernighan's heuristic which uses a definite favorable l-change (see the definition given below) requires excessive computation for ATSPs. Without having any definite favorable l-change, the algorithm is designed to be able to stop if there is no favorable change at every pair of steps. The computational results show that 90-city ATSPs can be solved in 4.43 minutes on a PRIME 400 minicomputer.

Below, we present the definitions of the *l-change* and the *primary change* for a given tour t, which are related to the *sequential primary change*.

Definition (*l-change of a tour t*):

> Let t and t' be ATSP tours, X and Y be disjoint sets of edges such that $t' = (t-X) \cup Y$. If $|X|=|Y|=l$, we say that t' is a l-change of t. A graph $G(t,t')$ is defined in such a way that, given the nodes of $G(t,t')$ corresponding to edges in X, if $x_i=(k, l)$ and $x_j=(p, q)$, $x_i, x_j \in X$, then there is an arc (x_i, x_j) in $G(t,t')$ iff $(k, q) \in Y$. For example, a tour t' in Figure 2.8(b) is a 5-change of the tour t of Figure 9(a), and Figure 9(c) displays the graph $G(t,t')$.

Definition (*Primary change*):

> A tour t' is a primary change if $G(t,t')$ consists of a single cycle. Figure 2.8(b), for instance, shows a primary change for the tour of Figure 2.8(a), but Figure 2.8(d) displays a non-primary change.

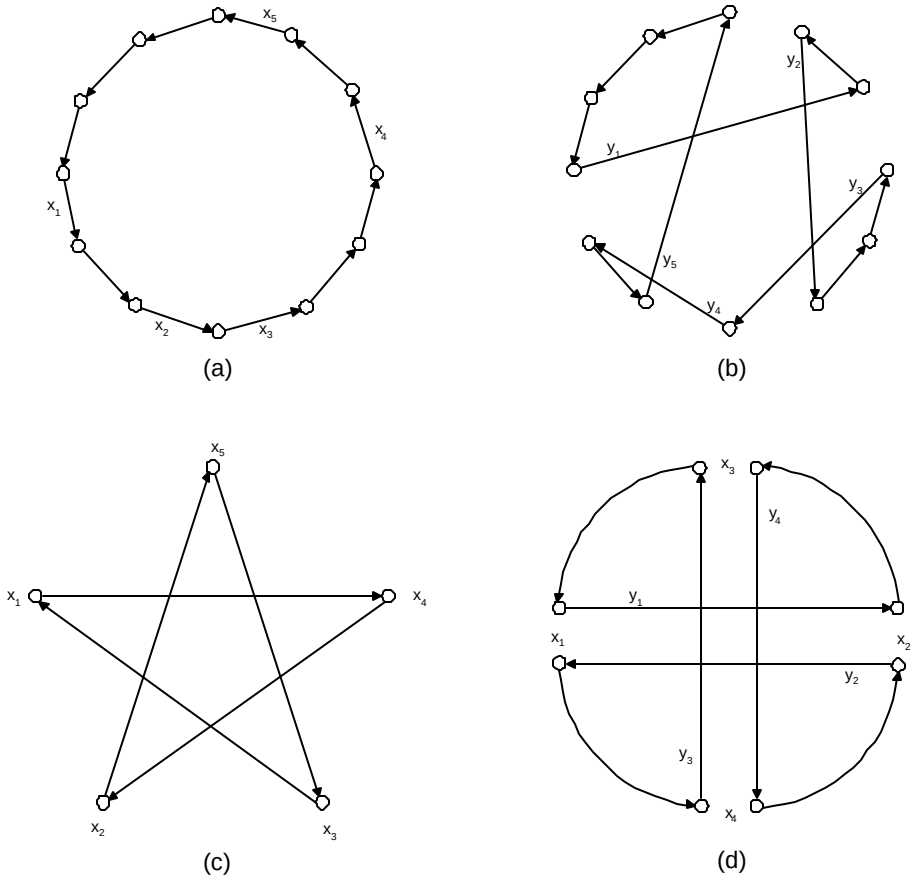


Figure 2.8 Definition of primary changes.

There are two ways to gradually construct a primary change t:

Decide x_1 to be removed (see Figure 2.9(a)).

The choice of y_1 (or generally y_{2i+1}) uniquely determines an x_{2i+2} and a closed cycle C_1 (Figure 2.9(b)).

Here we have two alternatives.

3-1) Choose y_2 to break the cycle C_1 and end up where the tour started (Figure 2.9(c)).

3-2) Choose y_2 to generate a new cycle C_2 (Figure 2.9(d)) and later break both cycles C_1 and C_2 (Figure 2.9(e)).

Accordingly, we can define the *sequential primary change* as follows:

Definition (*Sequential primary change*):

> If a *primary change* can be applied in a sequence, each immediately followed by the breaking of a cycle, it is called a s*equential primary change* (see Figure 2.9(c)).

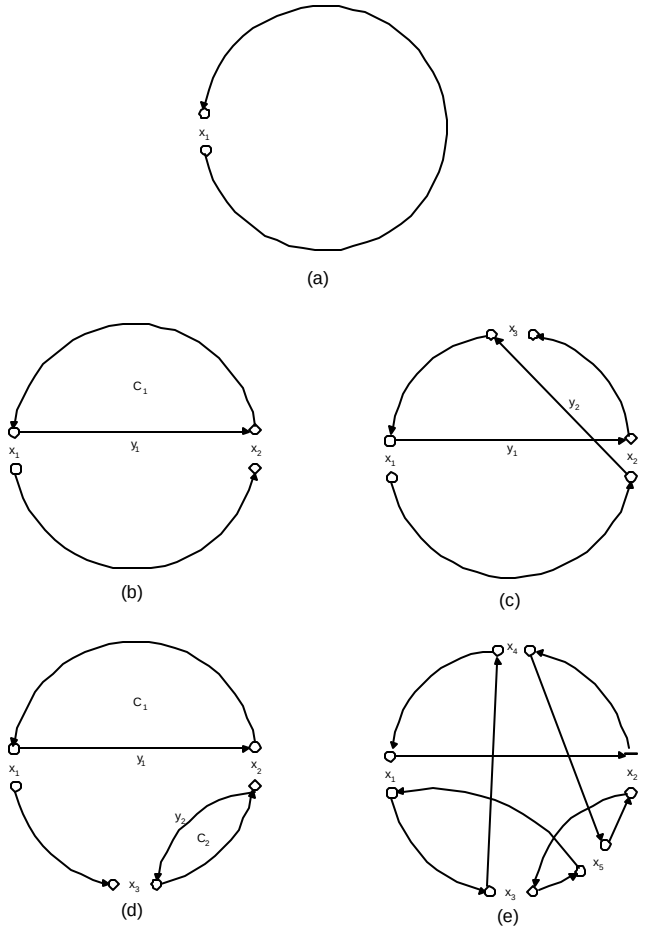


Figure 2.9 Construction of primary changes.

The heuristic for ATSP starts with an initial feasible tour t. It picks an edge x_1 of t, and tries to find a sequential primary change. If it fails, it backtracks with respect to the same x_1, and considers all possible y_1's in increasing order and y_2's. (Here, all y_2's that break the cycle created by y_1 are searched and the one that produces the most favorable 3-change is selected.) If it fails again, it backtracks with respect to all possible x_1's which it examines in decreasing order. This part of the procedure is detailed as follows:

Step 1 (*Initialization*): Set G_0=0, G^*=0, i^*=0, and i=1. Pick x_1=(k, l) as the largest cost edge.
Step 2 (*Pair of sub-steps*):
Sub-step 1 (pick y_i and create a new cycle C_i):
Let x_i=(k, l). Pick y_i=(k, q) subject to F and G_i. This determines x_{i+1} and generate a cycle C_i. If no such choice, stop.
Sub-step 2 (pick y_{i+1} and break the cycle C_i):
Let x_i=(p, q). Pick y_{i+1}=(p, r) subject to F and G_{i+1}. This determines x_{i+2} and a y^* closing the tour, and it breaks C_i. If no such choice, stop.
Step 3 (*Update*): $SP = G^*$. If $G_t{}^* > G^*$ then, $G^*=G_t{}^*$, delete x_{i^*+1}, …, x_{i+1} from current tour t, and add y_{i^*+1}, …, y_{i+1} to the tour t and set $i^*=i$+1.
Step 4 (*Stopping Criteria*): If $SP<G_{i+1}$, then set $i=i$+1, start a new pair of steps, else stop.

Where, F (Feasibility): The x's have not been y's in previous steps of the current search, and y_{i+1} breaks C_i.
G_i (working gain) = $G_{i-1} + c(x_i) - c(y_i)$.
$G_{i+1} = G_i + c(x_{i+1}) - c(y_{i+1})$.
$G_t{}^* = \sum_{j=1}^{i+2} c(x_j) - \sum_{j=1}^{i+1} c(y_j) - c(y^*) > 0$.
G^* (the best definite gain achieved so far at the step i^*).

2.1.4.5 Heuristic Approaches to the Asymmetric Traveling Salesman Problem with Time-windows (ATSP-TW)

As part of a branch-and-cut algorithm to solve ATSP-TW, Ascheuer et al. (1999) apply a series of heuristics. A sorting heuristic, a nearest-feasible-neighbor heuristic, and an insertion heuristic are applied for constructing an initial feasible tour. A swap heuristic, a two-node-exchange heuristic, a node-reinsertion heuristic, an arc-reinsertion heuristic, an arc-reversal heuristic, and the Or-exchange heuristic are applied for improving a current tour. For the convenience of implementation, an additional dummy node n+1 which denotes the depot is created such that $i \prec n+1 \ \forall i \in V$. (Here, "$\prec$" denotes the precedence relationship. For example, $i \prec j$ means that i has to precede j.) Ascheuer et al. also suggest an efficient order to implement these heuristics as follows:

Do{ construction heuristics with following order:
Sorting heuristic
Nearest-feasible-neighbor heuristic
Insertion heuristic 1
Insertion heuristic 2
}
If no feasible sequence found Stop.
Do{ improvement heuristics with following order:
Or-exchange heuristic

Arc-reversal heuristic
Swap heuristic
Arc-reinsertion heuristic
Node-reinsertion heuristic
Two-node exchange heuristic
} until no further improvement is achieved.

A. Tour Building Procedures

The various types of heuristics used for tour building are as follows:

1) Sorting heuristic:

Check if the trivial sequence $(0, 1,\ldots, n\text{-}1, n, n\text{+}1)$ is feasible.

Sort the nodes according to increasing r_i and check whether this sequence is feasible.

Sort the nodes according to increasing d_j and check whether this sequence is feasible.

Sort the nodes according to the midpoints of the time-windows $m_j = e_j+(e_j+l_j)/2$ and check whether this sequence is feasible.

2) Nearest-feasible-neighbor heuristic:

For each feasible arc $(0, i)\in A$,

Enlarge the current subtour $(0, v_1, v_2,\ldots,v_k)$ by an arc (v_k, v_l) resulting in the smallest increase in the objective value and guaranteeing feasibility.

3) Insertion heuristic:

Construct the initial subtour $P'=(0, v_1,\ldots, v_k, n\text{+}1)$ by finding the shortest path from 0 to n+1.

Enlarge the current subtour P' by choosing a node j satisfying one of following criteria.

b1) Among all unsequenced nodes $V'=V\backslash\{v_1,\ldots, v_k\}$, choose the node $j\in V'$ that yields the lowest increase in the path length, i.e., $d_{min}(j)=\min\{d_{min}(i)|i\in V'\}$, where $d_{min}(i) = \min\{ w_{v_l i} + w_{i v_{l+1}} - w_{v_l v_{l+1}} \mid i\in V', v_l\in P'$ and subtour $(0, v_1,\ldots, v_l, i, v_{l+1},\ldots, v_k, n\text{+}1)$ is feasible$\}$.

b2) Among all unsequenced nodes $V'=V\backslash\{v_1,\ldots, v_k\}$, choose the node $j\in W$ that has the lowest number of feasible insertion positions and insert this nodes at the cheapest of these positions.

B. Improvement Heuristics

The details of the improvement heuristics are as follows:

1) Swap heuristic:

Given a feasible tour $T=(v_0, v_1,\ldots, v_{n+1})$, construct a new tour T by scanning through the current tour and checking whether swapping two subsequent nodes v_i and v_{i+1}, i =1,..,n-1, results in a feasible solution with a better objective value. If a better solution is found, this new tour is accepted. The procedure is repeated until no further improvement is achieved.

2) Two-node-exchange heuristic:

Given a feasible tour $T=(v_0, v_1,\ldots, v_{n+1})$, construct a new tour T by exchanging any two nodes (not only subsequent nodes) in the current tour. If a better solution is found, this new tour is accepted. This procedure is repeated until no further improvement is achieved.

3) Node-reinsertion heuristic:

Given a feasible tour $T=(v_0, v_1,\ldots, v_i, v_j, v_k,\ldots, v_{n+1})$, construct a subtour $T'=(v_0, v_1,\ldots, v_i, v_k,\ldots, v_{n+1})$ by eliminating an inner node v_j. Try to reinsert v_j in the best position in T' such that the new tour T is feasible. If a better solution is found, this new tour is accepted. This procedure is repeated until no further improvement is achieved.

4) Arc-reinsertion heuristic:

Given a feasible tour $T=(v_0, v_1,\ldots, v_i, v_j, v_k, v_l,\ldots, v_{n+1})$, construct a subtour $T'=(v_0, v_1,\ldots, v_i, v_l,\ldots, v_{n+1})$ by eliminating two consecutive nodes v_j and v_k. Try to reinsert the arc (v_j, v_k) at any position in T' such that the new tour T is feasible. If a better solution is found, this new tour is accepted. This procedure is repeated until no further improvement is achieved.

5) Arc-reversal heuristic:

Given a feasible tour $T=(v_0,\ldots, v_j, v_k,\ldots,v_l, v_m,\ldots, v_{n+1})$, construct a tour $T=(v_0,\ldots, v_j, v_l,\ldots,v_k, v_m,\ldots, v_{n+1})$ by reversing the subpath $(v_k,\ldots,v_l)$ such that the new tour T is feasible. If a better solution is found, this new tour is accepted. This procedure is repeated until no further improvement is achieved.

6) Or-exchange heuristic:

Given a feasible tour $T=(v_0, v_1,\ldots, v_i,,\ldots, v_j,\ldots, v_{n+1})$, remove the subpath $(v_i,\ldots,v_j)$ from the current tour T, and try to reinsert it between any two subsequent nodes v_l and v_{l+1} such that the new tour T is feasible. If a better solution is found, this new tour is accepted. This procedure is repeated until no further improvement is achieved.

2.2 Literature Review on Network Assignment Problem (NAP)

The NAP problem on the taxiway has many similarities with urban transportation network problems. In urban networks, NAP solutions provide the best routes for drivers traveling to their own destinations so that the system (or drivers) can reach an optimal (or equilibrium) state. In solving urban transportation network problems, three types of sub-components are generally involved:

- Network assignment algorithm which, using the shortest path information, allocates traffic demand, i.e., vehicles, on the routes,
- Shortest path algorithm which, using the link travel times, provides the shortest paths for a given O-D pairs on the network, and
- Link performance function(s) which represent the relationship between the traffic volume on a link and the travel time along the link.

Finding an adequate link performance function is considered as a difficult task, and is still a controversial issue. The difficulty arises from the dynamic feature of traffic conditions. Some research groups [Leonard et al. 1978, Van Aerde 1985, 1999] use a simplified version of link performance function having two components: free running times along the link and delays at junction [Rakha, 1990]. Other groups [Peeta and Mahmassani 1995, Mahmassani, 1998] use simulation models as a tool for measuring link travel time rather than using a closed-formed link performance function. Some researches [Ran et al. 1997] suggest more sophisticated functions which are capable of capturing the dynamic characteristic of vehicle behaviors. In this research, we assume that the link travel time is a function of only delays at a junction. It is further assumed that the delays at taxiway junctions are proportional to the number of conflicting vehicles.

Among those three sub-components, we review the previous studies on the network assignment algorithm and the shortest path algorithm which have been well defined and successfully studied.

2.2.1 Literature Review on Network Assignment Algorithms

Depending on the time-dependency, network assignment algorithms can be classified into two types: 1) static network assignment algorithm, and 2) dynamic network assignment algorithms.

2.2.1.1 Static Assignment Algorithm

In 1952, Wardrop established two mutually independent network assignment principles for the static assignment problem. According to the first principle, users on the network choose a route that minimizes their own travel time. In the second principle, users distribute themselves on the network in such a way that the average (or marginal) travel time for all users for each route from origin to destination is equal so that the aggregate vehicle-hours spent in traveling is minimized. These two assignment principles are also-called "*User Equilibrium* (*UE*)" and "*System optimal* (*SO*)" respectively. Two critical conditions arise from two static network assignment principles [Papacostas and Prevedouros, 1993]:

1) *User equilibrium* (*UE*) is the state where no traveler can improve his/her travel time by unilaterally changing routes. (In other words, for each O-D pair, the travel time on all used paths is equal, and also less than or equal to the travel time that would be experienced by a single vehicle on any unused path.)
2) *System optimal* (*SO*) is the state where the travelers cannot improve the total system travel time by jointly changing routes in any fashion. (In other words, for each O-D pair, the marginal travel time on all used paths is equal, and also less than or equal to the marginal travel time that would be experienced by a single vehicle on any unused path.)

Finding *UE* (or *SO*) solutions is a well-researched problem and various techniques are commonly used in urban transportation studies [Sheffi, 1985]. The basic notation adopted in this research project is shown below:

x_a^n: flow on link a at n^{th} iteration,
t_a: travel time on link a at n^{th} iteration,
q_{rs}: trip rate between origin r and destination s,
$t_a()$: link travel time function for link a.

A) All-or-nothing assignment

The all-or-nothing assignment method is one in which the entire flow for any given O-D pair *r-s*, q_{rs}, is assigned to the minimum-travel-time path connecting this pair. The usual steps for the all-or-nothing assignment are:

Step 0: *initialization.* Perform all or nothing assignment based on the shortest paths for all trips obtained by using $t_a^0 = t_a(0)$ for all the links. Obtain link flows $\{x_a^0\}$ for all the links. Set $n = 1$.

Step 1: *update the link travel times*. Set $t_a^n = t_a(x_a^{n-1})$.

Step 2: *network loading*. Assign all the trips to their shortest paths on the network using the all-or-nothing strategy based on the travel times $\{t_a^n\}$. Obtain the link flows $\{x_a^n\}$ for all the links.

Step 3: *convergence test.* If $\max_a\{| x_a^n - x_a^{n-1} |\} \le k$, then stop. Otherwise, set $n \leftarrow n + 1$ and go to Step 1.

Considering how all-or-nothing assigns all of q_{rs} to the shortest path, the algorithm is not successful at converging to the state of user equilibrium. To overcome this problem, the following algorithms have been devised by the Federal Highway Administration (FHWA).

Step 0: *initialization.* Perform all-or-nothing assignment based on the shortest paths for all trips obtained by using $t_a^0 = t_a(0)$ for all the links. Obtain the link flows $\{x_a^0\}$ for all the links. Set $n = 1$.

Step 1: *update and smooth the link travel times*. Set $t_a^n = t_a(x_a^{n-1})$ and $t_a^n = 0.75\ t_a^{n-1} + 0.25 t_a^n$ for all the links.

Step 2: *network loading*. Perform an all-or-nothing assignment based on the travel times $\{t_a^n\}$. Obtain the link flows $\{x_a^n\}$ for all the links.

Step 3: *stopping rule*. If $n = N$, go to Step 4. Otherwise, set $n \leftarrow n + 1$ and go to Step 1.

Step 4: *averaging*. Set $x_a^* = \frac{1}{4}\sum_{i=0}^{3} x_a^{n-i}$, find the link travel times, $t_a^* = t_a(x_a^*)$ for all the links and stop. (Here, $\{x_a^*\}$ approximate the link flows at equilibrium.)

There are two modifications used above. First, the link travel times are updated by smoothing the link travel times from both the current and the previous iterations using certain weighting factors. Second, the final link flows are obtained by averaging link flows from the last four iterations. These modifications have proven to be somewhat helpful in obtaining solutions that are closer to a true equilibrium state.

B) Incremental Assignment

In the incremental assignment method, the flows for a given O-D pair are assigned as a packet which represents a portion of the origin-destination matrix at each iteration. The travel times are updated based on the total traffic flows assigned to the links. The stepwise procedure for this approach is outlined below.

Step 0: *initialization.* Divide each origin-destination demand into N equal portions (i.e. set $q_{rs}^{n} = q_{rs}/N$). Perform an all-or-nothing assignment based on $t_a^0 = t_a(0)$. Obtain a set of link flows $\{x_a^0\}$. Set $n = 1$ and $x_a^0 = 0$.

Step 1: *update link travel times*. Set $t_a^n = t_a(x_a^{n-1})$.

Step 2: *incremental network loading.* Assign q_{rs}^n to the network based on the travel times $\{t_a^n\}$ for all origin-destination (i.e. rs) pairs. Obtain a set of link flows $\{w_a^n\}$ from this assignment, where w_a^n is the flow on link a resulting from the assignment of the nth increment of O-D matrix on to the network.

Step 3: *flow summation.* Set $x_a^n = x_a^{n-1} + w_a^n$ for all the links.

Step 4: *stopping rule*. If $n = N$, stop with the current link flows as the prescribed solution. Otherwise, set $n \leftarrow n + 1$ and go to Step 1.

One important fact in the incremental assignment procedure is that as the number of increments, N, grows, the condition for *UE* can be achieved more closely. This point provides us with the rationale for using an incremental assignment strategy in microscopic simulation models. In microscopic simulation models, the vehicles are treated as individual objects, which means that each vehicle is assigned one by one rather than as a packet of size q_{rs}/N on the network. In this case, the number of increments (N) is the same as the total number of vehicles, and this might tend to produce the *UE* state.

C) Method of Successive Averages (MSA)

The MSA uses the following basic algorithm step of most minimization procedures.

$$\boldsymbol{x}^{n+1} = \boldsymbol{x}^{n} + \alpha_n \cdot \boldsymbol{d}^{n}$$

where, $\boldsymbol{x}^n$ is the link flows at nth iteration,

α_n is a step size and

$\boldsymbol{d}^n$ is a descent direction vector computed at $\boldsymbol{x}^n$.

In MSA, the step size α_n (n=1,2,…) is not determined on basis of some characteristics of the current solution but determined a priori. For this method to converge, some conditions of the objective function and α_n have to be satisfied. These conditions include twice differentiability of the objective function, $\sum_{n=1}^{\infty} \alpha_n = \infty$ and $\sum_{n=1}^{\infty} \alpha_n^2 = \infty$ (see Sheffi (1985) for a complete discussion). The method is outlined below.

Step 0: *initialization*. Perform an all-or-nothing assignment based on $t_a^0 = t_a(0)$. Obtain a set of link flows $\{x_a^1\}$. Set $n = 1$.

step 1: *update*. Set $t_a^n = t_a(x_a^n)$.

Step 2: *direction finding*. Assign all the trips to the network based on the travel times $\{t_a^n\}$. Obtain a set of auxiliary link flows $\{y_a^n\}$ from this assignment.

Step 3: *move*. Find the new flow pattern for all the links by setting $x_a^{n+1} = x_a^n + (1/n)(y_a^n - x_a^n)$.

Step 4: *convergence test*. If a convergence criterion is met, stop with the current solution, $\{x_a^{n+1}\}$, as the set of prescribed (near equilibrium) link flows. Otherwise, set $n \leftarrow n + 1$ and go to Step 1.

D) Convex Combination Method

Without using a predefined step size α_n, the convex combination method finds α_n at each iteration by solving a Non-linear Programming (NLP) problem which is subject to linear constraints. The resulting NLP problem can be replaced by a much simpler linear approximation, and solved using the Frank-Wolfe (1956) algorithm. The method is outlined below.

Step 0: *initialization.* Perform an all-or-nothing assignment based on $t_a = t_a(0)$. Obtain a set of link flows $\{x_a^1\}$. Set $n = 1$.

Step 1: *update link travel times*. Set $t_a^n = t_a(x_a^n)$.

Step 2: *direction finding.* Assign all the trips to the network based on the travel times $\{t_a^n\}$. Obtain a set of auxiliary link flows $\{y_a^n\}$ from this assignment.

Step 3: *line search.* Find α_n by solving $\min_{0 \le \alpha \le 1} \sum_a \int_0^{x_a^n + \alpha (y_a^n - x_a^n)} t_a(w)\, dw$.

Step 4: *move.* Set $x_a^{n+1} = x_a^n + \alpha_n (y_a^n - x_a^n)$ for all the links.

Step 5: *convergence test.* If a convergence criterion is met, stop with the current solution, $\{x_a^{n+1}\}$ as the set of prescribed (near equilibrium) link flows. Otherwise, $n \leftarrow n + 1$ and go to Step 1.

Methods A) and B) are called heuristic (or sometimes called non-equilibrium) assignment methods in the sense that these may not converge to the equilibrium solution. On the other hand, method D) is formulated as a mathematical programming technique to achieve either *UE* and *SO*. It can be shown that solutions of mathematical formulations are consistent with to the conditions of *UE* and *SO* defined by Wardrop.

2.2.1.2 Dynamic Assignment Algorithm

Similar to the static assignment problem, there are two types of dynamic assignment problems: 1) *dynamic system optimal assignment problem* (*DSO*) which seeks to minimize the total system travel time over the planning horizon, and 2) *dynamic user equilibrium assignment problem* (*DUE*) which seeks time-dependent user path assignments that satisfy the temporal extension of Wardrop's *UE* condition [Peeta and Mahmassani, 1995].

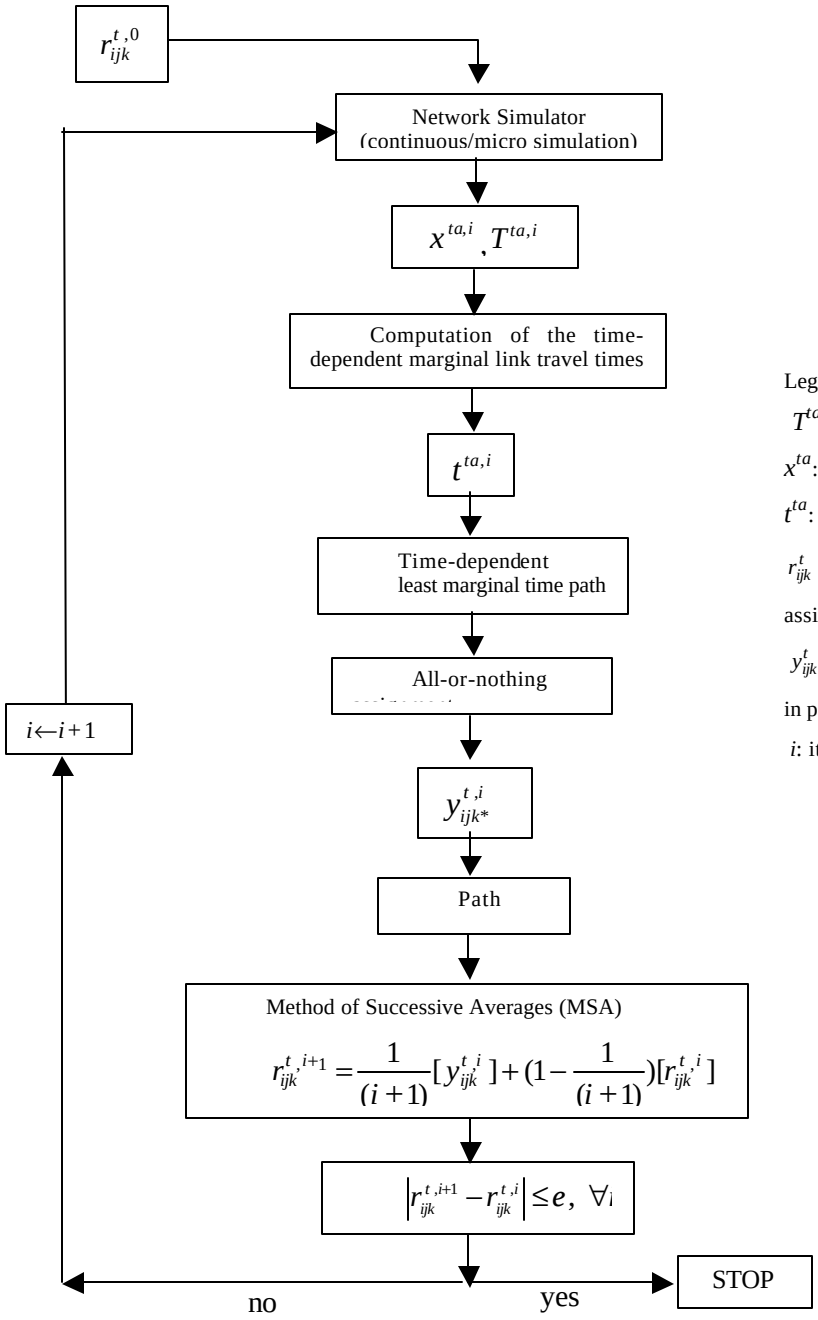
Janson and Robles (1995) define the *DUE* as follows: Given a set of zone-to-zone trip tables containing the number of vehicle trips from each origin zone in successive time intervals of 1 to 10 minutes each, determine the volumes of vehicles on each link in each time interval such that, for each O-D pair of zones, no path has a lower travel time than any used path for trips departing within a given time interval. Janson (1991) proves that the *DUE* condition for fixed departure times is a temporal generalization of Wardrop's condition for a static user equilibrium.

Using optimal control theory, Friesz et al. (1989) and Wie (1989) present formulations for dynamic traffic assignment in continuous time, in which the equilibrium condition is stated that

no used path between any two nodes must have a higher travel time than any other path at any instant. Ran et al. (1993) refine and extend optimal control models to include elastic demand and departure time choice in user equilibrium or system optimal forms. Friesz et al. (1993) formulate the simultaneous route choice and departure time problem in continuous time as a variational inequality problem. Ran et al. (1997) propose the time-dependent travel time functions for dynamic assignment on signalized network links which can be used to solve discrete-time dynamic assignment problems.

Peeta and Mahmassani (1995) point out that virtually all of previous *DUE* models have link flows as the decision variables, and that path flows obtained from link-based formulations are not always unique. They suggest a formulation for the path-based assignment problem which involves a non-explicit function of path travel times. Instead of defining the extremely complicated path travel time function, a simulator called "DYNASMART" is developed and used to evaluate experienced path travel times. By applying Lagrangian multipliers, Peeta and Mahmassani derive the conditions for a *Dynamic System Optimal* (*DSO*) state. At a *DSO state*, the time-dependent marginal travel times for all used paths connecting a given O-D pair are equal, and less than or equal to the time-dependent marginal travel times on any unused routes. In order to find marginal link travel times, a curve fitting method is suggested. As a solution algorithm, the method of successive averages (MSA) is used to determine the new path flows for the next iteration. Using time-dependent experienced link travel times measured by the simulator, the time-dependent shortest paths for all O-D pairs are computed. The complete algorithm is depicted in Figure 2.10.

In the context of the simulation model named INTEGRATION, Van Aerde (1985) suggests that a minimum path tree table indirectly constitutes a traffic assignment function, and forms the basis upon which vehicles make route selection decisions. Specifically, the minimum path tree table provides a list of the turning movements which correspond to the minimum paths for each destination. These minimum path turning movements identify all downstream links corresponding to a vehicle's minimum cost path, given the current location of the vehicle and its eventual destination. As drivers re-check these minimum path trees at each node, they automatically reselect new paths if previous paths become congested and/or competing alternative paths become faster. This path selection (and re-selection) process is based on real-time information and attempts to reproduce a continuous dynamic equilibrium.



Legend:

T^{ta}: travel time on link a during time interval t

x^{ta}: number of vehicles on link a during time interval t

t^{ta}: marginal travel time on link a during time interval t

r_{ijk}^{t} : number of vehicles departing from i to j in period t that are assigned to path k

y_{ijk*}^{t} : auxiliary number of vehicles on paths departing from i to j in period t that are assigned to path k

i: iteration counter.

Figure 2.10 Algorithmic Framework for Time-Dependent System Optimal Assignment Problem [Peeta and Mahamassani, 1995].

2.2.2 Shortest Path (SP) Algorithms

2.2.2.1 Importance of SP Algorithm in Transportation Studies

Finding shortest paths is a classical problem in the field of Operations Research. Recently, with the development of Integrated Transportation System (ITS), the time-dependent assignment problem for real-time traveler information system has captured some renewed attention. It is widely accepted that the successful implementation of real-time traffic network control system depends on the efficiency of SP algorithms. The following comments show the importance of the SP problem [Gallo, 1985].

> If there is one routine that is never absent from any computer code used in transportation analysis, it is certainly the SP routine. About 80 percent of computation time in traffic assignment is consumed in finding shortest paths.

2.2.2.2 Definitions of SP Problems

Depending on the time-dependency of link travel times and number of shortest paths to be found, SP problems are categorized into static SP, static k-SP, time-dependent SP, and time-dependent k-SP problems. In the static SP problem, the link costs (or travel times) are assumed to be independent of time. In the time-dependent SP problem, on the contrary, the link costs change over time. Instead of finding a single path for a given origin-destination (O-D) pair, the k-SP problem deals with multiple shortest paths. The k-SP problems are also classified as being either static or time-dependent. The definition of various SP problems are summarized in Table 2.3 [Subramanian, 1997]. The basic notation used in this section is shown below:

N: node set of a given graph $G(A, N)$,
A: arc set of a given graph $G(A, N)$.
A_T: arc set of a directed spanning tree T.
$FS(i)$: forward star of node i,
$RS(i)$: reverse star of node i,
c_{ij}: the travel cost (or travel time) of link $(i, j) \in A$,
$c_{ij}(t)$: the travel cost (or travel time) of link $(i, j) \in A$ departing node i at time t,
s: source node,
tn: terminal node,
i, j: intermediate node.

Table 2.3 Various Definitions of SP Problems.

Problem	Given	Find
Static SP	A graph $G(N, A)$ having $\|N\|$ nodes and $\|A\|$ arcs, and a distinguished source node s and a destination node t, and a set of link costs, c_{ij}, associated with each arc (i, j).	The shortest path from s to t.
Static k-SP	Same as static SP problem.	The first, second,…, k^{th} shortest paths from s to t, for any user-specified $k \in 1,2,...$
Time-dependent SP	A graph $G(N, A)$ having $\|N\|$ nodes and $\|A\|$ arcs, and a distinguished source node s and a destination node t, and a set of time-dependent link delays, $c_{ij}(t)$, associated with each arc (i, j).	The shortest path from s to t, starting from s at time $t=t_0$.
Time-dependent k-SP	Same as time-dependent SP problem.	The first, second,…, k^{th} shortest paths from s to t, starting from s at time $t=t_0$, for any user-specified $k \in 1,2,...$

2.2.2.3 Static SP Algorithm

2.2.2.3.1 Network Flow Programming Approach

The static SP problem can be formulated as a minimum-cost network flow problem (Bazaraa et al., 1990).

$$\underset{x}{\text{Minimize}} \quad \sum_{(i,j)\in A} c_{ij} x_{ij}$$

$$\text{subject to :} \quad \sum_{j\in FS(i)} x_{ij} - \sum_{j\in RS(i)} x_{ji} = s_i \qquad \forall i \in N$$

$$x_{ij} \geq 0 \qquad \forall (i, j) \in A$$

where $s_s = 1, s_{tn} = -1, s_i = 0 \quad \forall i \neq (s, tn)$

and where, s_s, s_{tn}, s_i: supply of node s, tn, i, respectively.

Due to the unimodularity property of this network structure, optimal extreme point solutions take only integral values.

2.2.2.3.2 Bellman's Principle of Optimality

Bellman introduced the well-known "Principle of Optimality" which provides the basic foundation for recursive algorithms including Dynamic Programming. Using the "Principle of Optimality", Bellman (1966) developed the following equation. Let us denote

$$c_{kj} = \begin{cases} \text{the (finite) length of arc } (k, j), \text{if there is such an arc.} \\ +\infty, \text{otherwise.} \end{cases}$$

f_j = the length of a shortest path from the origin to node j.

If there are no directed cycles with negative length, it is clear that $f_s = 0$, where the node s is the origin node. For each node $j, j \neq s$, there must be some final arc (k, j) in a shortest path from node s to j. Whatever the identity of k, it is certain that $f_j = f_k + c_{kj}$. This follows from the fact that the part of the path which extends to node k must be a shortest path from 1 to k. If this is not true, the overall path to j would not be as short as possible (this is the "Principle of Optimality"). The relationship between nodes s, k and j is shown in Figure 2.11.

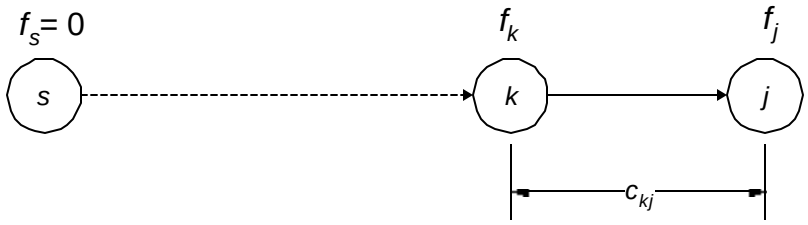


Figure 2.11 Bellman's Principle of Optimality.

The shortest path length must satisfy the following system of equations (called Bellman's equations):

$$f_s = 0 ,$$

$$f_j = \min_{k \neq j} \{ f_k + c_{kj} \}, \; j = 1, .., n .$$

2.2.2.3.3 Labeling Algorithm

Labeling algorithms are known as the most popular and efficient methods to solve the shortest path problem. The label in the algorithm represents the tentative shortest path length from the source node to that node. There are two types of labeling algorithms: label setting (LS) and label correcting (LC).

The LS algorithm sets the label of one node permanently at each iteration, thus increasing the shortest path vector by one component at each iteration. The LC algorithm does not set any label permanently. Instead, all the components of the shortest path vector are obtained simultaneously after the algorithm terminates. Some features of these two SP algorithms are summarized in Table 2.4.

Table 2.4 Comparison of LS and LC Algorithm (1).

Label setting (LS) algorithm	Label correcting (LC) algorithm
Designate one label as permanent at each iteration Applicable only to Acyclic networks and Non-negative arc lengths problem	All labels are temporary until the final step when all become permanent Applicable to all classes of problems
Both are iterative Assign tentative labels to nodes at each step (Label = the upper bound on the shortest path cost)	
LS Algorithm $\subseteq$ LC Algorithm	

The LC method always exchanges (augments, or updates) arcs in A_T in a manner that replaces or shortens the unique path from the source node s to v in T, where T is the directed spanning tree and A_T is the in T. However the LC method does not guarantee that the new path is a shortest path until termination occurs.

The LS algorithm has become known as Dijkstra's algorithm since Dijkstra was one of the first to discover it independently. This algorithm finds the shortest paths from the source node s to all other nodes in a network with nonnegative arc lengths. Table 2.5 shows generic pseudocodes for the LS and LC algorithms.

Let, f_i: travel distance (or travel time or label) to node i from source node s (which is an upper bound on the shortest length to node i),
c_{ij}: the (finite) length of arc (i,j),
S : permanent label set,
$\overline{S}$: temporary label set.

Table 2.5 Comparison of LS and LC Algorithm (2).

LS Algorithm (Dijkstra's Algorithm)	LC Algorithm
Node-based selection. Using permanent and temporary label.	Arc-based selection

$S=\{\}, \overline{S}=N$;
$f_s = 0$ and $predecessor(s) = 0$;
$f_i = \infty$ for each node $i \in N$;
 	
while $\lvert S\rvert \le n$ where $n=\lvert N\rvert$ do
select node $i \in \overline{S}$ which $f_i = \min\{f_j : j \in \overline{S}\}$;
$S = S \cup \{i\}$;
$\overline{S} = \overline{S} - \{i\}$;
for each (i, j), $j \in FS(i)$ do
if $f_j > f_i + c_{ij}$ then
$f_j = f_i + c_{ij}$;
$predecessor(j) = i$;
end
end
end
	$f_s = 0$, $predecessor(s)=0$;
	$f_i = \infty$ for each $i \in N$-$\{s\}$;
	
	while some arc(i,j) satisfies $f_j > f_i + c_{ij}$ do
	$f_j = f_i + c_{ij}$;
	$predecessor(j) = i$;
	end

2.2.2.4 Time-dependent SP Algorithm

Assuming that the time-dependent link costs for all links, $c_{ij}(t)$, are positive integer values, Cook and Halsey (1966) have extended Bellman's Principle of Optimality to solve a time-dependent SP algorithm. According to Cook and Halsey, the minimum time of travel to node *tn* starting from node *i* at time *t*, $f_i(t)$, is defined by the following functional equation and shown in Figure 2.12:

$$f_i(t) = \begin{cases} \min\limits_{j \neq i} \{c_{ij}(t) + f_j[t + c_{ij}(t)]\} & for\ i \neq tn; t \in S \\ 0 & for\ i = tn; t \in S \end{cases}$$

where, S: the discrete time set; $S=\{t_0, t_0+1, t_0+2, \ldots, t_0+T\}$,
T: the fixed upper-bound of travel time from node *i* to *tn*.

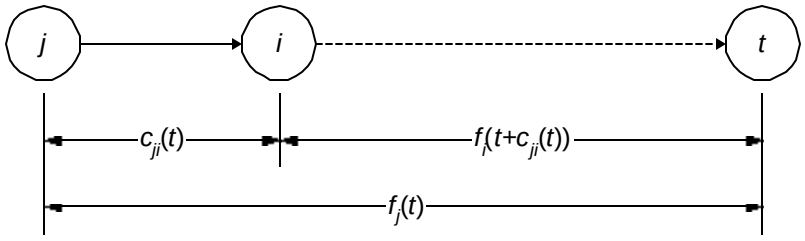


Figure 2.12 Time-dependent SP Algorithm [Cook and Halsey, 1985].

Dreyfus (1969) has suggested the use of Dijkstra's algorithm to determine time-dependent shortest paths where the link costs are any real-valued times. The minimum time of travel to node j starting from node s at time 0, f_j, is defined as follow:

$$f_j = \begin{cases} \min_{i \neq j} \{ f_i + c_{ij}(f_i) \} & for\ i \neq tn, \\ 0 & for\ i = tn \end{cases}$$

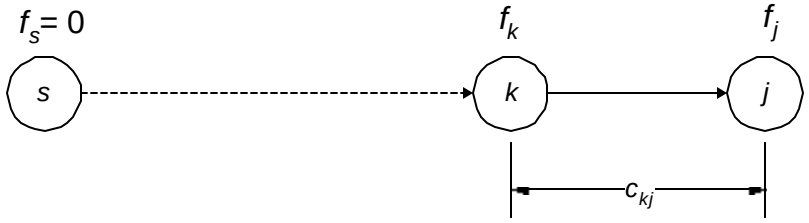


Figure 2.13 Time-dependent SP Algorithm [Dreyfus, 1969].

While Cook and Halsey's method applies the Principle of Optimality in forward form, Dreyfus's algorithm is implemented in backward fashion. Dreyfus's main algorithm is exactly the same as that of the LS algorithm except that time-dependent link costs are used in the optimality constraint. The following procedure finds the minimum path tree from node s to all nodes starting at 0.

$S=\{\}, \overline{S}$ =N; $f_i = \infty$ for each node $i \in N$; $f_s = 0$ and *predecessor*(s) = 0;
while $|S| \leq n$ do
  select $i \in \overline{S}$ which $f_i = \min\{f_j : j \in \overline{S}\}$;
  $S = S \cup \{i\}$;
  $\overline{S} = \overline{S} - \{i\}$;
  for each $(i, j), j \in FS(i)$ do
    if $f_j > f_i + c_{ij}(f_i)$ then

```
                $f_j = f_i + c_{ij}$;
                $predecessor(j) = i$;
            end
        end
    end
```

Harpern (1977) first noted the limitation of Dreyfus' approach and showed that if there exists a $y>0$ such that $y + c_{ij}(t + y) < c_{ij}(t)$, then the departure from node i must be delayed, or the optimal path might include cycles. Kaufman and Smith (1993) studied the assumptions under which the existing TDSP algorithms would work. To illustrate the point, consider the simple network shown in Figure 2.14. The resulting SP from 1 to 4 starting at time 0 is 1-3-4, with the total path cost $f_4 = 15$.

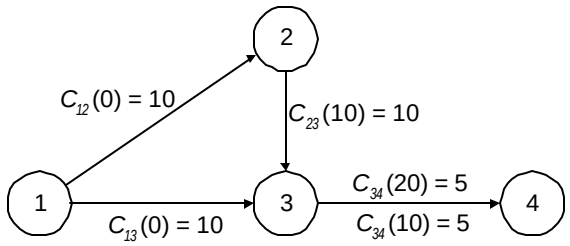


Figure 2.14 Example of TDSP (1).

Now, let us assume that as shown in Figure 2.15, the travel time for link (3,4) at time 10 increases to 20. Then the shortest path for the trip from 1 to 4 starting at time 0 is 1-2-3-4 with a travel cost of 25. Here, it should be noticed that the driver who enters link (3,4) at time 20 can finish the trip at 25 but another one who enters the same link at time 10 cannot finish the trip until 30. This result is unreasonable in general transportation networks because the first-in-first-out (FIFO) condition is violated.

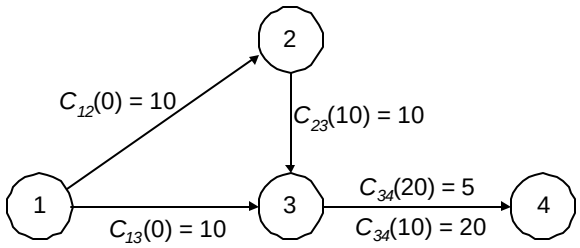


Figure 2.15 Example of TDSP (2).

Kaufman and Smith make a consistency assumption preventing the time-dependent link cost to indicate passing as follow.

For any arc $(i,j) \in A$, $t_1 + c_{ij}(t_1) \leq t_2 + c_{ij}(t_2)$ for all $t_1, t_2 \in T$ such that $t_1 \leq t_2$.

i.e., $\dfrac{c_{ij}(t_1) - c_{ij}(t_2)}{t_2 - t_1} \leq 1$ for $t_1 \leq t_2$.

They also show that under the assumption that the link-delay function follows the first-in-first-out (FIFO) rule or consistency assumption, any static LS or LC algorithm can be extended to the time-dependent case (using the time-space network formulation).

Orda and Rom (1990) studied various types of waiting-at-nodes scenarios and proposed algorithms for these different cases. They showed that if waiting is allowed at nodes (UW), then the consistency assumption is not required. They prescribed an algorithm for identifying optimal waiting times at the source node if waiting is not allowed elsewhere in the network. Furthermore, they demonstrated that for the forbidden waiting case, the paths obtained without the consistency assumption may not be simple, and showed that the continuous-time version of the problem is NP-Hard. Table 2.6 shows the three cases of time-dependent SP algorithms studied by Orda and Rom. Sherali, Ozbay, and Subramanian (1998) prove NP-Hardness of various versions of time-dependent shortest path problems, and develop efficient solution algorithms.

Table 2.6 Three Types of Time-dependent SP Algorithms.

UW (Unrestricted Waiting)	Vehicles may wait an unlimited duration at any nodes
SW (Source Waiting)	Vehicles may wait an unlimited duration only at source nodes
FW (Forbidden Waiting)	Vehicles are not permitted to wait at any nodes

For the UW case, the suggested solution algorithm matches that of Dreyfus', except that $D_{ij}(t)$ is defined as follows.

$$D_{ij}(t) = w + c_{ij}(w+t)$$

where, $D_{ij}(t)$: total link travel time from i to j
w : waiting time at node i
$c_{ij}(w+t)$: travel cost from node i to j starting at $(w+t)$

Figure 2.16 illustrates the graphical derivation of $D_{ij}(t)$.

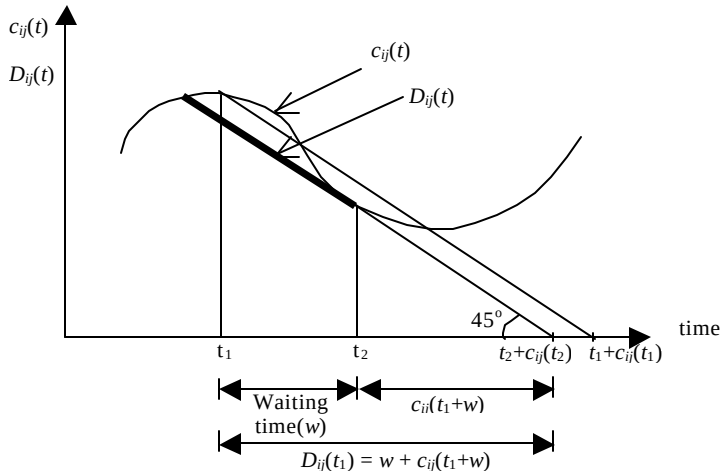


Figure 2.16 Total Link Travel Time for the UW Problem.

2.2.2.5 Implementation Issues

Gallo and Pallottino (1985) point out that the traditional classification of shortest path algorithms into LC and LS is somewhat unsatisfactory because of its dependence on the behavior of the algorithm rather than on their data structure. Dijkstra's algorithm is a type of LS algorithm where the arc lengths are non-negative, while it becomes a LC algorithm if there are some negative arc lengths in the graph. Gallo and Pallottino suggest that it is more desirable to classify the SP algorithms based on the data structure which is the way to keep so-called "candidate nodes" for next iteration. Let T be a directed spanning tree of G rooted at node s, and d_v be the length of the unique path in T from s to v, $v \in G$. Then T is a shortest path tree with origin s ($T = T(s)$) if and only if the following condition holds:

$$f_i + c_{ij} - f_j \geq 0 \qquad \text{for all } (i, j) \in A \qquad (2.1)$$

Then all the arc-based shortest path algorithms (i.e., LC SP algorithms) can be stated as having the following procedures:

Step 1: Initiate a directed tree T rooted at r and for each $v \in N$, let f_v be the length of the path from s to v in T.

Step 2: Let $(i, j) \in A$ be an arc for which condition (2.1) is not satisfied, then adjust the vector f by setting $f_j = f_i + c_{ij}$, and update the tree T replacing the current arc incident into node j by the new arc (i,j).

Step 3: Repeat Step 2 until condition (3.1) is satisfied for all $(i, j) \in A$.

The important point in the implementation of this procedure is how to select an arc at Step 2 in order to check whether condition (2.1) is satisfied. Since $n < m$ ($n = |N|$, and $m = |A|$), it seems reasonable to select nodes rather than arcs. Once a node i is selected, condition (3.1) is checked on one or more (possibly all) arcs of forward stars of i, FS(i). (In the majority of the algorithms, all the arcs corresponding to the selected node's forward stars are checked once.) A general implementation of procedure for node-based LC SP algorithms is as follows:

Step 1: (Initialize) $f_s = 0$, $predecessor(s)=0$; $f_i = \infty$ for each $i \in N-\{s\}$; Q = $\{s\}$;
Step 2: (Select and update)
  Select $i \in$ Q; Q = Q - $\{i\}$;
  For each (i,j), $j \in FS(i)$ such that $f_j > f_i + c_{ij}$, do
    $f_j = f_i + c_{ij}$;
    $predecessor(j) = i$;
    Q = Q + $\{j\}$;
Step 3: (Iteration) if Q $\neq \varnothing$ then go to Step 2, else stop.

where, Q: a set of candidates nodes (or a list of scan eligible (SE)).

The initial tree at Step 1 is a star-shaped tree, with one dummy arc (s,i) for each $i \in N-\{s\}$. These dummy arcs are assigned a length equal to ∞. It is very important how to select the node i from the set of candidate nodes Q. In fact, almost all the practical shortest path algorithms are derived by properly defining the rule of selection and the particular data structure which is used to implement the set Q. Theoretically speaking, the time for node selection is bounded by $O(n^2)$ and the time for distance updates is bounded by $O(m)$. If the network is sparse (i.e., $n^2 >> m$), then the former time dominates the later. So, we need to reduce the node selection time without substantially increasing the time for updating the distances.

Figure 2.17 shows several types of data structures relevant to the selection schemes. Internal operations for each type of Queue is summarized in Table 2.7. A *sorted queue* is used for the LS SP algorithm in which a sorting method is imbedded so that the node having the least label can be selected from the set of candidate nodes. However, it should be noted that if the problem size is large, the sorting algorithm is not inexpensive in terms of computational cost.

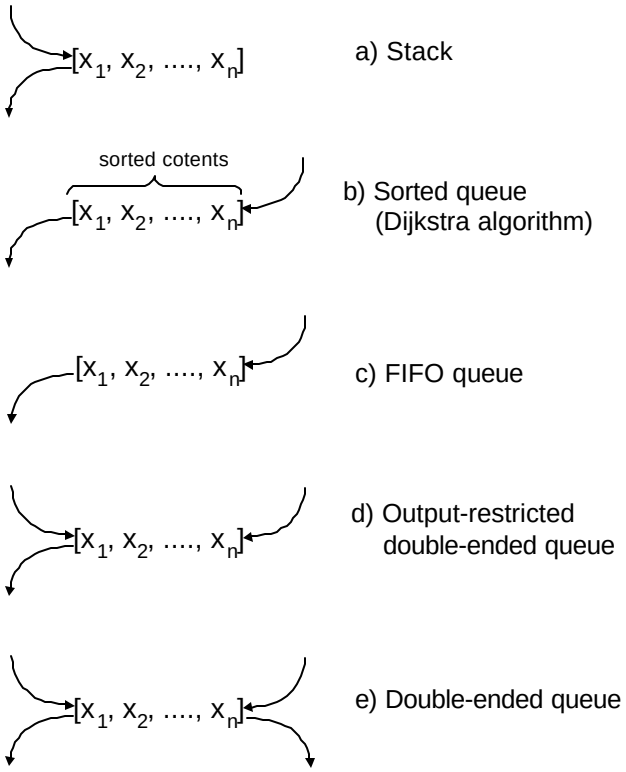


Figure 2.17 Types of Queues for Node Selection Schemes [reproduced from Tarjan, 1983].

To revise this drawback, the so called *double-ended queue* (DEQueue, see Figure 2.17 (d)) has been developed, which combines the properties of both the queue and the stack. In the DEQueue structure, the first time a node is to be inserted into the tail of the queue. When, later on, the same node again becomes a candidate node after being removed from the queue, it is inserted at the head of the queue.

Table 2.7 Operations for Various Queue Types.

	Sorted Queue	FIFO Queue	Output-restricted double-ended Queue	Double-ended Queue
enqueueFirst (=push)	×	×		
enqueueLast (=inject)				
dequeueFirst (=pop)				
dequeueLast (=eject)	×	×	×	
SortContents		×	×	×

On the other hand, the candidate nodes are always removed from the head of the queue. The rationale for using the DEQueue is that every time f_j is updated, except the first time, it is worth trying to decrease the labels of the successors of j in current tree. Table 2.8 summarizes the details of LC algorithms according to the data structures employed. Maintaining the DEQueue to handle the candidate nodes, Ziliakopoulos and Mahmassani (1993) devise the time-dependent shortest path algorithm. Figure 2.18 presents the pseudo code for this algorithm.

```
Call CREATE;

Call INSERT(N);
Current_Node = N; // to-node i

Do 1, While (SE list is not empty)

        Call DELETE(Current_Node);

        Do 2, for (All nodes J that can be directly reach Current_Node)
   // J is the reverse star of node i

                Next_Node = J;
                In_SE_List? = No;

                Do 3, for(t=1,M)

                        //Travel Time (j,i) starting at t
                        Current_Travel_Time = TRAVEL_TIME(Next_Node, Current_Node, t);

                        // TT(j,N) at t = TT(i,N) at (t+TT (j,i)) + TT(i,N) at t
                        New_Label = Label(Current_Node, t+Current_Travel_Time)
                                + Current_Travel_Time;

                        If( Label(Next_Node,t) ≤ New_Label ) then
                                Label(Next_Node, t) = New_Label;
                                In_SE_List? = Yes;
                                Path_Pointer(Next_Node, t, 1) = Current_Node;
                                Path_Pointer(Next_Node, t , 2) = t + Current_travel_Time;
                        Endif

3               Continue

                If (In_SE_List? = Yes) Call INSERT(Next_Node);
2       Continue
1 Continue

Procedure CREATE
        Do, for(Node=1,N-1), Deque(Node) = 0;
        Deque(N) = 999999;
        First = N;
        Last = N;

Procedure INSERT(Node)
        If(Deque(Node) = 0) then
                Deque(Last) = Node;
                Last = Node;
                Deque(Node) = 999999;
        Else if (Deque(Node) = -1) then
                Deque(Node) = First;
                First = Node;
        Endif

Procedure DELETE(Current_Node)
        Current_Node = First;
        First = Deque(Current_Node);
        Deque(Current_Node) ;
```

Figure 2.18 Pseudo Code for a Time-dependent SP Algorithm

[Adapted from Ziliakopoulos and Mahmassani, 1993].

Table 2.8 Comparison of Data Structures for LC Algorithms.

Generic LC Algorithm (Arc-based Selection)	Modified LC Algorithm (Node-based Selection)	
	Using FIFO Queue	Using Double Ended Queue
	Rationale: number of nodes << number of arcs.	Rationale: Every time f_j is updated, except the first time, it is worth trying to decrease the labels of the successors of j in current tree.

$f_s = 0$, $pred(s)=0$;
$f_j = \infty$ for each $j \in N-\{s\}$;
 	
while some arc(i,j) satisfies $f_j > f_i + c_{ij}$ do
$f_j = f_i + c_{ij}$;
$pred(j) = i$;
end
	$f_s = 0$, $pred(s)=0$;
	$f_j = \infty$ for each $j \in N-\{s\}$;
	FIFOQueue=$\{s\}$
	
	While FIFOQueue $\neq \{\}$ do
	Remove i from the head of FIFOQueue;
	For any (i,j), $j \in FS(i)$ do
	If $f_j > f_i + c_{ij}$ then{
	$f_j = f_i + c_{ij}$;
	$pred(j) = i$;
	if $j \notin$ FIFOQueue
	then add node i to the tail of FIFOQueue;
	}
	end
	end
		$f_s = 0$, $pred(s)=0$;
		$f_j = \infty$ for each $j \in N-\{s\}$;
		DEQueue=$\{s\}$
		
		While DEQueue $\neq\{\}$ do
		Remove i from the head of DEQueue;
		For any (i,j), $j \in FS(i)$ do
		If $f_j > f_i + c_{ij}$ then{
		if $j \notin$ FIFOQueue{
		if $f_j=\infty$ then
		insert j into the tail of DEQueue;
		else insert j into the head of DEQueue;
		}
		$f_j = f_i + c_{ij}$;
		$pred(j) = i$;
		}
		end
		end

2.3 Literature Review on Simulation Model

2.3.1 Types of Simulation Models

Simulation models can be classified according to the following general categories [Lieberman and Rathi, 1992]:

Discrete and continuous simulation models,
Microscopic, mesoscopic, and macroscopic simulation models, and
Deterministic and stochastic simulation models.

Discrete simulation models represent a system by asserting that the states of the system elements change abruptly at points in time. In contrast, continuous simulation models represent the system by changing state variables continuously over time [Law and Kelton, 1991]. Typically, continuous simulation models involve differential equations giving relationships for the rates of change of the state variables with time. If the differential equation is simple enough to be solved analytically, the solution provides the values of the state variables at any given time as a function of the values of the state variables at time zero. Because continuous models frequently are not tractable using an analytical approach, numerical analysis techniques, e.g., Runge-Kutta integration, are used to integrate the differential equations. For this reason, regardless of the nature of the real system which might be either discrete or continuous, two types of discrete simulation models are applied in practice: 1) discrete time simulation and 2) discrete event simulation models. For systems of limited size entities whose states change infrequently, discrete event simulation models are more appropriate in the sense of computational execution time. However, for systems where most entities experience a continuous change in state and where the model objectives require very detailed descriptions, discrete time models are likely to be the better choice [Lieberman and Rathi, 1992].

Traffic simulation models may be classified according to the level of detail with which they represent the system to be studied: 1) Microscopic, 2) Mesoscopic and 3) Macroscopic. A microscopic model deals with both system entities and their interactions at a high level of detail. A mesoscopic model generally represents entities at a higher level of detail but describes their activities and interactions at a much lower level of detail. On the other hand, a macroscopic model describes entities and their activities at a low level of detail. In a mesoscopic model, for example, the lane-changing maneuver could be represented for individual vehicles as an

instantaneous event with the decision based on the relative lane densities, rather than on detailed vehicle interactions. In macroscopic models, however, the traffic stream may be represented in some aggregate manner such as speed, flow, and density, and lane change maneuvers would probably not be represented.

In deterministic models, there are no random variables. In other words, all interactions between entities are fixed in the sense that relationships are defined by mathematical, statistical or logical equations. Stochastic models involve processes which include probability functions. The car-following model, for instance, may be modeled either as a deterministic or a stochastic problem by defining the driver's reaction time as a constant estimated value or as a random variable, respectively.

2.3.2 Previous Air Traffic Simulation Models

Since the early seventies, the FAA has developed computer simulation models to analyze airport operations. Contrasting with the analytic models which consist of a series of equations using fixed input parameters, most aviation and airport simulation models are discrete event, stochastic models which emulate the movements of aircraft on the airfield as well as in the airspace. In general, the simulation models produce the following statistics:

- Hourly runway, taxiway, and gate capacity
- Hourly and daily delays, travel times, flow rates, and queueing data etc.
- Annual delay and annual delay costs
- Annual delay savings computed from annual delay costs

ADSIM (Airfield Delay Simulation Model, 1976)

ADSIM is a microscopic, discrete event, stochastic simulation model, and known as one of the most detailed models to evaluate the operations and sources of delay on the airport's surface http://www.tc.faa.gov/act500/capacity/modelsq.htm]. ADSIM simulates the movement of aircraft on the airport surface and in the immediate airspace. An airport is composed of a common approach and departure corridors. The principal inputs to the model include aircraft routings, runway and taxiway usage, runway occupancy times and exit probabilities, aircraft approach and taxiing velocities, aircraft separations, gate service times, aircraft demand and mix, and ATC rules and procedures. Outputs of the model are hourly arrival and departure flow rates, travel times, and arrival and departure delays for each runway. Also provided are total delays for each

link on the airfield, departure queue lengths, and individual aircraft delays. ADSIM has a capability to represent the movements in animation mode.

RDSIM (Runway Delay Simulation Model)

Developed mainly for runway capacity and delay analyses, RDSIM simulates operations on runways and generates information on both capacities and delays [http://www.tc.faa.gov/act500/capacity/modelsq.htm]. This model is a discrete event, stochastic simulation model. During the simulation, it is assumed that arrival and departure demands are uniformly distributed. The model simulates runway operations with arrival-priority, departure-priority, or balanced arrivals and departures. This model can be used to compute runway capacity at an acceptable level of delay and maximum runway throughput. Compared with ADSIM, RDSIM requires less detailed inputs and less computational effort. The inputs consist of runway usage, runway occupancy times, exit probabilities, aircraft demand and mix, aircraft approach velocities, aircraft separations, and ATC rules and procedures. The outputs include delay statistics with graphics showing delay versus demand along with other operational details.

SIMMOD (The airspace and airfield model)

SIMMOD is the first model to analyze the complex airspace interactions between airports. SIMMOD satisfies the need to analyze delays, capacity and fuel consumption resulting from changes in airspace utilization and operational procedures beyond an airport's immediate airspace [SIMMOD3 Simulation Module, 1993]. In SIMMOD, each aircraft's movement is traced individually and ATC required the actions for aircraft operations are also simulated. Inputs for SIMMOD are traffic demand and fleet mix, route structures (both in the airspace and on the airport surface), runway use configurations, separation rules and control procedures, aircraft performance characteristics in airspace and airfield links, airspace sectorization, interactions among multiple airports, and weather conditions. SIMMOD uses a link-node structure to represent the gate/taxiway and runway/airspace route system. Input parameters depending on aircraft type include the permissible airborne speed ranges for use by ATC, runway occupancy times, safety separations, landing roll and takeoff characteristics, taxi speeds, and runway/taxiway utilization. Gate utilization depends on aircraft type and airline. The output from SIMMOD reports statistics about individual aircraft delay, travel time, and fuel consumption as well. A simulation log containing information on various simulated events are also generated at the user's request.

2.3.3 Aircraft-Following Models

If there is more than one aircraft moving on the same taxiing path and the vehicles are close enough to interact with each other, the leading aircraft behavior affects the following aircraft. To represent this situation, aircraft-following models are introduced. The main concepts governing aircraft-following models are borrowed from well-researched issues in transportation studies concerning car-following models. There are several types of car-following models readily applicable to simulation practices. It should be noticed that there is an important assumption about the speed-acceleration relation applied to all types of vehicle-following models. In this analysis, the vehicle's ability to accelerate is assumed to decreases linearly as a function of speed.

Distance-controlled vehicle-following model: The distance-controlled logic for vehicle-following models is based on the assumption that the acceleration of the following vehicle at time $t+\Delta t$ is decided by the distance between the leading and the following vehicle. The acceleration for the following vehicle is decided by the equation given below (Starfield, 1990).

$$\ddot{x}_{n+1}^{t+\Delta t} = k[(x_n^t - x_{n+1}^t) - D],$$

if $\ddot{x}_{n+1}^{t+\Delta t} > \ddot{x}_{\max}$ then $\ddot{x}_{n+1}^{t+\Delta t} = \ddot{x}_{\max}$, and if $\ddot{x}_{n+1}^{t+\Delta t} < \ddot{x}_{\min}$ then $\ddot{x}_{n+1}^{t+\Delta t} = \ddot{x}_{\min}$,

where, $\ddot{x}_{n+1}^{t+\Delta t}$: acceleration for aircraft $n+1$ at time $t+\Delta t$,

$(x_n^t - x_{n+1}^t)$: distance between aircraft n and $n+1$ at time t,

k = design parameter,

D = safety distance.

Speed-controlled vehicle-following model: In the speed-controlled logic, the acceleration of the following vehicle at time $t+\Delta t$ is modeled by the speed difference between the leading and the following vehicle. Mathematically this can be expressed as,

$$\ddot{x}_{n+1}^{t+\Delta t} = k(\dot{x}_n^t - \dot{x}_{n+1}^t),$$

if $\ddot{x}_{n+1}^{t+\Delta t} > \ddot{x}_{\max}$ then $\ddot{x}_{n+1}^{t+\Delta t} = \ddot{x}_{\max}$, and if $\ddot{x}_{n+1}^{t+\Delta t} < \ddot{x}_{\min}$ then $\ddot{x}_{n+1}^{t+\Delta t} = \ddot{x}_{\min}$,

where, $\ddot{x}_{n+1}^{t+\Delta t}$: acceleration for aircraft $n+1$ at time $t+\Delta t$,

$(x_n^t - x_{n+1}^t)$: speed difference aircraft vehicle n and $n+1$ at time t,

k = design parameter.

Generalized vehicle-following model: The generalized car-following model proposed by Gazis et al. (1961) assumes that the acceleration of the following vehicle at time $t+\Delta t$ is influenced by three dependent variables: the differences in both distance and the speed between the leading and the following vehicles, and following vehicle's speed. The generalized car-following model can be expressed as follows:

$$\ddot{x}_{n+1}^{t+\Delta t} = a \frac{(\dot{x}_{n+1}^{t+\Delta t})^m}{(x_n^t - x_{n+1}^t)^l}(\dot{x}_n^t - \dot{x}_{n+1}^t), \qquad (2.2)$$

if $\ddot{x}_{n+1}^{t+\Delta t} > \ddot{x}_{\max}$ then $\ddot{x}_{n+1}^{t+\Delta t} = \ddot{x}_{\max}$, and if $\ddot{x}_{n+1}^{t+\Delta t} < \ddot{x}_{\min}$ then $\ddot{x}_{n+1}^{t+\Delta t} = \ddot{x}_{\min}$,

where, α, m, l: design parameters.

It should be noted that by integrating the equation (2.2) and applying the boundary conditions at the steady state, the generalized vehicle-following model can be related to the traffic stream model. Also, the resulting macroscopic models are diverse depending on the values of l and m. For example, in the case that l =0, m =0, which is the speed-controlled car-following model, the resulting macroscopic model is $q = a\ (1 - k/k_j)$, $a = q_m$, where q = flow, k = density, k_j = jam density and q_m = maximum flow. The various macroscopic models are shown in Table 2.9.

2.3.4 Data Structures for Network Representation

The performance of a network algorithm depends not only on the algorithm itself, but also on the manner used to represent the network within a computer. By representing a network more cleverly and by using improved data and list structures, we can often improve the running time of an algorithm [Ahuja et al., 1993].

Table 2.9 Macroscopic Models from Vehicle-following Models (Gerlough and Huber, 1975).

	$m = 0$		$m = 1$	
$l = 0$	$q = a\ (1 - k/k_j)$ *i)	$a = q_m$ *ii)	-	-
1	$q = a\ k\ ln(k_j/k)$ *iii)	$a = u_m$ *iv)	-	-
3/2	$q = a\ (1 - (k/k_j)^{1/2})$ *v)	$a = u_f$	-	-
2	$q = a\ k\ (1 - k/k_j)$ *vi)	$a = u_f$	$q = a\ k\ e^{(k/k0)}$ *vii)	$a = u_f, k_0 = k_m$
3	-	-	$q = a\ k\ e^{(k/k0)} - ½(k/k_0)$ *viii)	$a = q_m$

*i) Chandler, Herman, and Montroll (1958), *ii) Pipes (1953), *iii) Greenberg (1959)
*iv) Gazis (1961), *v) Drew (1965), *vi) Greenshiled (1934), *vii) Edie (1961)
*viii) Drake, Schoefer, and May (1961)

In representing a network, we need to store two types of information: (1) the network topology; that is, the network's node and arc structures, and (2) attribute data such as costs (C_{ij}) and capacities associated with arcs. There are several ways to represent a weighted graph $G=(N,A)$, $|N|=n$, $|A|=m$. Various graph representation methods including their advantages and disadvantages are reviewed with a simple network shown in Figure 2.19.

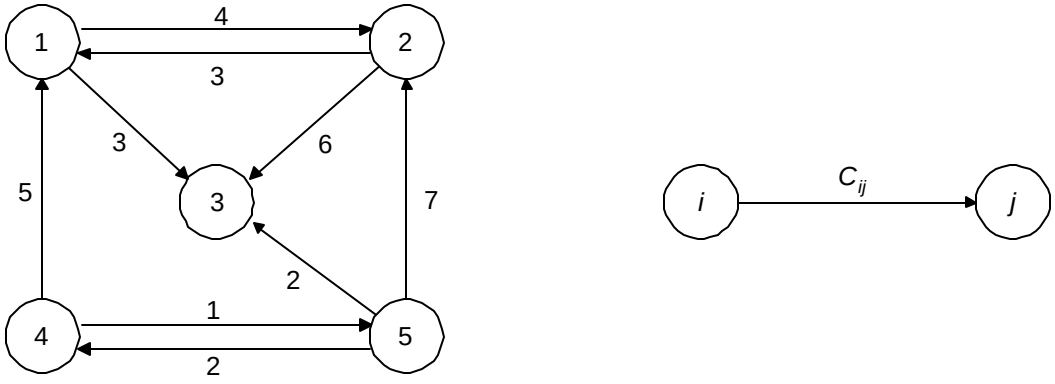


Figure 2.19 An Example Graph with 5 Nodes and 9 Arcs (i.e., n=5, m=9).

(Node-Arc) Incident matrix: Construct an $n \times m$ matrix which contains one row for each node and one column for each arc. The column corresponding to each arc (i,j) has only two non-zero elements: It has a +1 in the row corresponding to node i and a -1 in the row corresponding to node j (see the Figure 2.20). Separate $n \times m$ matrices should be generated for data storage.

Advantages: An incident matrix can be used as a constraint matrix of the minimum cost flow problem. This matrix possesses several important theoretical properties such as total unimodularity, etc. (see Bazaraa et al. (1990) for details).

Disadvantages: The incident matrix which has ($n \times m$-2m) zeros is inefficient in storage space.

$node \backslash arc$	(1,2)	(1,3)	(2,1)	(2,3)	(4,1)	(4,5)	(5,2)	(5,3)	(5,4)
1	1	1	−1	0	-1	0	0	0	0
2	-1	0	1	1	0	0	-1	0	0
3	0	−1	0	-1	0	0	0	-1	0
4	0	0	0	0	1	1	0	0	-1
5	0	0	0	0	0	-1	1	1	1

Figure 2.20 Node-Arc Incident Matrix for the Example Network.

Node-node adjacency matrix: Construct an $n \times n$ matrix which has a row and column corresponding to every node. The ij^{th} entry of the matrix equals 1 if arc $(i,j) \in A$ and equals 0 otherwise.

Advantages: This matrix is space efficient if the network is sufficiently dense and the simplicity of the matrix allows us to implement the network algorithm easily.

Disadvantages: Needs another $n \times n$ matrix for data representation. An identification of outgoing/emanating arcs of a node is in time proportional to n. In sparse networks this may be a bottleneck operation for an algorithm.

from node \ to node	1	2	3	4	5
1	0	1	1	0	0
2	1	0	1	0	0
3	0	0	0	0	0
4	1	0	0	0	1
5	0	1	1	1	0

Figure 2.21 Node-Node Adjacency Matrix for the Example Network.

Adjacency lists: The arc adjacency list $A(i)$ of a node i is defined as the set of arcs emanating from that node, that is, the set of arcs $(i,j) \in A$ obtained as j ranges over the nodes of the network. Similarly, the node adjacency list of node i is defined as the set of nodes j for which $(i,j) \in A$. The adjacency list representation stores the node adjacency list of each node as a singly linked list. To implement this list, n linked lists, one for each node, should be generated. Array pointers that point to the first cell of each linked list, pt(i), are also constructed.

Advantages : Adjacency list representations are relatively efficient in storage. Deletion and addition of nodes can be done in constant time.

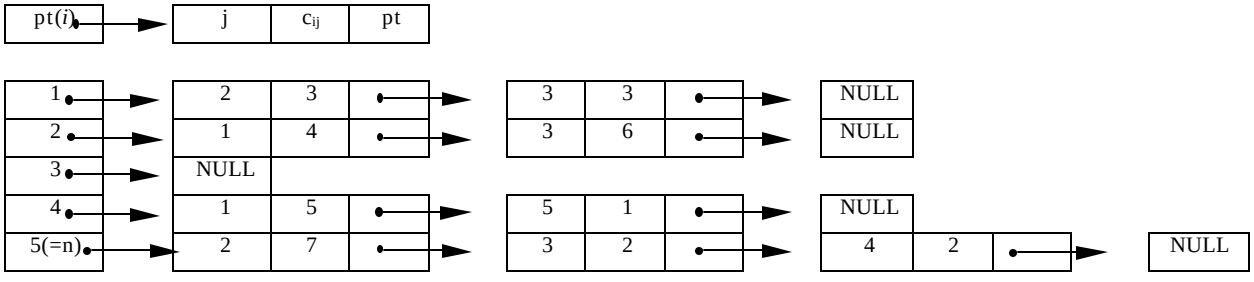
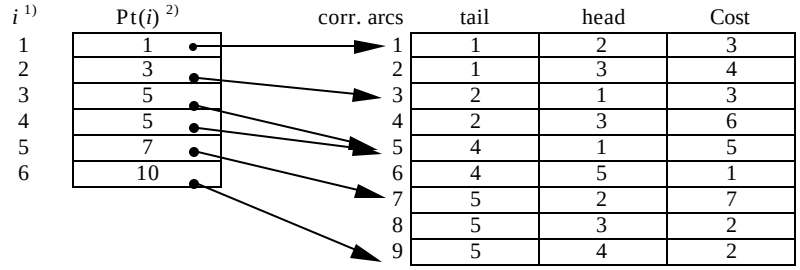


Figure 2.22 Adjacency List Representations for the Example Network.

Forward/reverse star representation: The forward star representation of a network is similar to the adjacency list representation in the sense that it also stores the node adjacency list for each node. However, instead of maintaining these lists as linked lists, it stores them in several arrays: tail(.), head(.), and cost(.). Pointers pointing to the first element of the corresponding arc list are stored in the pointer array pt(.). By convention: a) if the forward star of a node i is empty pt(i) = pt(i+1); b) pt(n+1) = m+1. Thus, information related to the emanating arcs (i.e., forward star) of a node i is stored in tail(.), head(.) and cost(.) from position pt(i) to pt(i+1)-1.

Advantages: This representation is more space efficient than the adjacency list representations and can be applied in general computer languages like FORTRAN.

Disadvantages: Addition/deletion of a node requires time proportional to m which can be time consuming.



i [1]	Pt(i) [2]	corr. arcs	tail	head	Cost
1	1	1	1	2	3
2	3	2	1	3	4
3	5	3	2	1	3
4	5	4	2	3	6
5	7	5	4	1	5
6	10	6	4	5	1
		7	5	2	7
		8	5	3	2
		9	5	4	2

[1] i is the from nodes, the pointer array contains pointers to the first elements of the corresponding arc lists.

Figure 2.23 Forward Star Representation for the Example Network.

Chapter 3. Model Development for ASP

3.1 Assumptions for Ready-times

In order to obtain a sequence from ASP, we need to make some assumptions regarding the departing aircraft ready-times. In fact, there are two types of ready-times: the ready-time to pushback from a gate, and the ready-time for takeoff on the runway. The first one is the time when the aircraft completes its boarding process, and waits for taxiing-out instructions from a ground (or a ramp) controller. Initially, this time can be obtained from the flight schedule. However, as airport operations progress over time, flight delays appear and new ready-times can be obtained from the airline operations center.

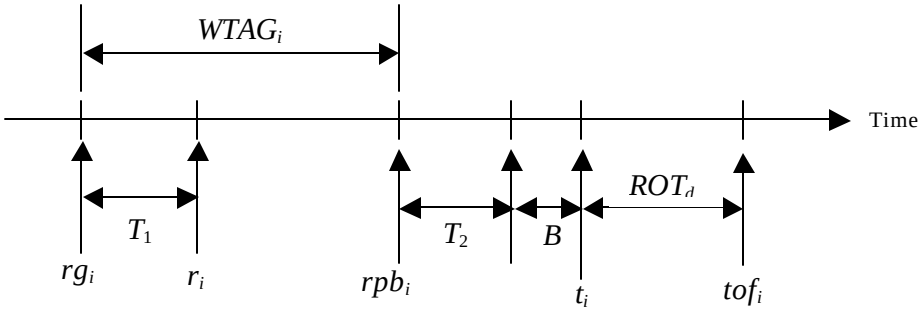
The second ready-time is the instance at which the aircraft can begin the takeoff roll on the runway. The earliest ready-time for takeoff (or so called "nominal takeoff time") is estimated as the ready-time to pushback from the gate plus the taxiing time from the gate to the runway departure queue. In our formulation, the ready-times to takeoff, r_i, are used as one set of the input data to solve the ASP problem.

One interesting aspect of the problem is that once we have the takeoff times as prescribed by ASP, we can suggest pushback times from the gates in a manner that reduces congestion both on the taxiway network and around departure queues. The recommended push-back times can be computed as depicted in Figure 3.1. Mathematically, the relationship between ready-time instances for a departing aircraft within the time horizon is given by,

> Recommended pushback time from the gate
>
> = Optimal takeoff time – (Taxiing time + Buffer time).
>
> (Here, buffer time includes expected communication time and estimated waiting time in the departure queue.)

As seen in Figure 3.1, we can make different uses of the taxiing time (T), depending on the type of computation being performed. T_1 is the taxiing time interval that estimates the earliest takeoff ready-time (r_i) required for ASP. Initially, this time can be obtained from historical data. (This data can be found in the Consolidated Operations and Delay Analysis System (CODAS) database which includes estimated taxiing times on ground networks at busy airports.) On the other hand, T_2 is the actual taxiing time to reach the takeoff ready-time from the recommended pushback time

(rpb_i). This time can be obtained from a Network Assignment Problem (NAP) which computes the optimal route and taxiing time for each aircraft after model ASP prescribes the optimal takeoff times (t_i).



Where,

rg_i : ready-time at gate
r_i : earliest ready-time to takeoff
rpb_i : recommended ready-time to push-back
t_i : optimal time for takeoff
tof_i : finishing time for takeoff
T_1 : regular taxiing time from the gate to the departure queue
T_2: taxiing time to obtain the recommend push-back time
B : buffer time (i.e., waiting time in departure queue)
ROT_d: runway occupancy time
$WTAG_i$: waiting time at gate.

Figure 3.1 Time Relationships for the Departing Aircraft.

3.2 First-come-first-serve (FCFS) Approaches

When air traffic controllers in the control tower direct landing and departing aircraft, they generally use a first-come-first-serve (FCFS) strategy, giving priority to landing aircraft. This strategy provides a sequencing of the aircraft along with completion times, total delays, and so on. This solution can be used to compare the performance of manual (FCFS) and optimal sequencing methods.

To consider both arrival and departure operations simultaneously, minimum separation rules should be defined for all possible aircraft group combinations. Table 3.1 shows the minimum separation rules for all the cases considered in our analysis.

Table 3.1 Minimum Separation (seconds).

departure→ departure case.

Leading \ Following	Heavy	Large	Small
Heavy	60	90	120
Large	60	60	90
Small	60	60	60

departure→ arrival case.

Leading \ Following	Heavy	Large	Small
Heavy	50	53	65
Large	50	53	65
Small	50	53	65

arrival → departure case.

Leading \ Following	Heavy	Large	Small
Heavy	40	40	40
Large	35	35	35
Small	30	30	30

arrival → arrival case.

Leading \ Following	Heavy	Large	Small
Heavy	99	133	196
Large	74	107	131
Small	74	80	98

The runway occupancy time (ROT) is computed for every operation in order to assess the processing times. ROT is defined as the time between the instance when an aircraft crosses the

runway threshold and the instance when the same aircraft clears the imaginary plane of the runway at a turnoff. These times are specified in Table 3.2.

Table 3.2 Runway Occupancy Times (seconds).

Operation \ Aircraft Type	Heavy	Large	Small
Arriving	40	35	30
Departing	50	40	30

Using the foregoing data set, the FCFS sequence delays can be calculated for all aircraft operations. An illustrative example using a randomly generated flight schedule is shown in Table 3.3. In this example, a due-time is obtained by adding a maximum delay of 600 seconds to all ready-times.

Table 3.3 Aircraft Schedule (Randomly Generated).

Max Delay: 600 (seconds)

Order	Arrival				Departure			
	Flight ID	Aircraft Type	Nominal Touchdown Time		Flight ID	Aircraft Type	Nominal Takeoff Roll Time	
			Ready	Due			Ready	Due
1	A1	S	98	698	D1	H	32	632
2	A2	L	164	764	D2	H	364	964
3	A3	L	205	805	D3	L	409	1009
4	A4	L	268	868	D4	H	542	1142
5	A5	H	494	1094	D5	S	571	1171
6	A6	H	550	1150	D6	S	643	1243
7	A7	L	643	1243	D7	L	744	1344
8	A8	L	834	1434	D8	L	881	1481
9	A9	L	962	1562	D9	S	955	1555
10	A10	L	973	1573	D10	H	996	1596

In FCFS with a full landing priority strategy, arriving aircraft should not be delayed by departing flights. In other words, the arriving aircraft can be delayed only to resolve arrival conflicts. The first step for FCFS sequencing is to obtain the arrival times by checking if there is any conflicting arrival. The following steps show how to calculate the times for arrivals to cross runway threshold, considering the minimum separation rules between two consecutive landing aircraft in the FCFS sequence.

fcfs touchdown time for 1st landing aircraft = nominal ready-time for 1st landing aircraft

for i = 2 to (total number of arriving aircraft)
    if [(fcfs touchdown time for $(i-1)^{th}$ aircraft + minimum separation time between the $(i-1)^{th}$ and i^{th} aircraft) > nominal ready-time of i^{th} aircraft]
        fcfs touchdown time for i^{th} aircraft =
            fcfs touchdown time for $(i-1)^{th}$ aircraft
            + min. separation time between the $(i-1)^{th}$ and i^{th} aircraft
    else
        fcfs touchdown time for i^{th} aircraft =
            nominal ready-time for i^{th} aircraft
end

The resulting schedule for arrivals is shown in Table 3.4.

Table 3.4 First-come-first-serve Sequence for Landing Aircraft.

Order	Arrival				
	Flight Number	Aircraft Type	Nominal Touchdown Time		FCFS Touchdown Time
			Ready	Due	
1	A1	S	98	698	98
2	A2	L	164	764	178
3	A3	L	205	805	285
4	A4	L	268	868	392
5	A5	H	494	1094	523
6	A6	H	550	1150	621
7	A7	L	643	1243	701
8	A8	L	834	1434	834
9	A9	L	962	1562	962
10	A10	L	973	1573	1069

If the inter-arrival time between two consecutive aircraft is sufficient for one or more departures, these are scheduled. Otherwise, departing aircraft are delayed until a slot having a sufficient time to allow a departure is found. Table 3.5 presents the final result of the FCFS sequence with landing priority.

Table 3.5 First-come-first-serve[1] Sequence (with Landing Priority).

(Seconds)

Arrival							Departure						
Seq	FLT No.	Acft Type	S_T [2]	ROT[3]	C_T [4]	Delay [5]	Seq	FLT No.	Acft Type	S_T	ROT	C_T	Delay
							1	D1	H	32	40	72	0
2	A1	S	98	30	128	0							
3	A2	L	178	40	218	14							
4	A3	L	285	40	325	80							
5	A4	L	392	40	432	124							
							6	D2	H	432	40	472	68
7	A5	H	523	30	553	29							
8	A6	H	621	30	651	71							
9	A7	L	701	40	741	58							
							10	D3	L	736	35	771	327
11	A8	L	834	40	874	0							
							12	D4	H	874	40	914	332
13	A9	L	962	40	1002	0							
							14	D5	S	992	30	1022	421
15	A10	L	1069	40	1109	96							
							16	D6	S	1109	30	1139	466
							17	D7	L	1164	35	1199	420
							18	D8	L	1224	35	1259	343
							19	D9	S	1319	30	1349	364
							20	D10	H	1369	40	1409	373
Total Delay: 3586 (sec), Completion Time: 1408 (Sec.)													

[1] In FCFS with landing priority, the landing aircraft should not be delayed later than its FCFS ready-time.

[2] S_T: Starting time

[3] ROT: Runway Occupancy Time

[4] C_T: Completion Time

[5] Delay = FCFS operation time – Nominal Ready-time.

3.3 Integer Programming Approaches

3.3.1 Formulation

Starting with known preliminary information about the aircraft such as aircraft types, nominal arrival/departure times, maximum delay times, minimum separation rules, etc., the problem can be formulated as a combinatorial optimization problem. The following is a mixed-integer programming model for a single runway or closely spaced runways where arrivals and departures are dependent.

Minimize z

subject to

$$t_i + \sum_{\substack{j \neq i \\ j \in J_0}} p_{ij} x_{ij} \leq z \qquad \forall\, i \in J \qquad (3.1)$$

$$r_i - t_i \leq 0 \qquad \forall\, i \in J \qquad (3.2)$$

$$t_i - d_i \leq 0 \qquad \forall\, i \in J \qquad (3.3)$$

$$t_j \geq t_i + p_{ij} - (1 - x_{ij})(d_i - r_j + p_{ij}) \qquad \forall\, i \in J_0, j \in J_0, j \neq i \qquad (3.4)$$

$$\sum_{\substack{j \neq i \\ j \in J_0}} x_{ij} = 1 \qquad \forall\, i \in J_0 \qquad (3.5)$$

$$\sum_{\substack{i \neq j \\ i \in J_0}} x_{ij} = 1 \qquad \forall\, j \in J_0 \qquad (3.6)$$

$$x_{ij} + x_{ji} \leq 1 \qquad \forall\, i \in J_0, j \in J_0, j \neq i \qquad (3.7)$$

$$t_0 = 0,\ t_i \geq 0 \qquad \forall\, i \in J \qquad (3.8)$$

$$x_{ij} \in \{0,1\} \qquad \forall\, i \in J_0, j \in J_0, j \neq i \qquad (3.9)$$

where,

$J = \{1, \ldots, n\}$: A set of n aircraft operating (i.e., landing or departing) on a single runway

$J_0 = J \cup \{0\}$: Aircraft 0 is an imaginary aircraft which has 0 ready-time (i.e., r_0=0) and 0 due-time (i.e., d_0=0)

x_{ij} : 1 if aircraft i directly precedes aircraft j, and 0 otherwise

x_{0i} : 1, if aircraft i is the first in the sequence, and 0 otherwise

x_{i0} : 1, if aircraft i is the last in the sequence, and 0 otherwise

t_i : the start time of aircraft i (i.e., time for touchdown or takeoff)

d_i : the due-time for aircraft i

r_i : the ready-time for aircraft i,

p_{ij} : p_{i0}= runway occupancy time of aircraft i (i.e., ROT_i) for $\forall i \in J$.
p_{0j}= setup time for the first aircraft j.
else if $i \neq j$, $i \neq 0$, $j \neq 0$, then p_{ij} = minimum separation time between the leading aircraft i and the following aircraft j.
if $i=j$, $p_{ij}=0$.

Constraint (3.1) states that the objective function value is no less than the last aircraft's completion time. Ready and due-times constraints are considered in constraints (3.2) and (3.3). The minimum separation rules are enforced by constraint (3.4). Two types of assignment constraints are represented by (3.5) and (3.6). Constraint (3.7) is a (two-city) sub-tour elimination constraint which prevents the resulting sequence from having any sub-tour consisting of two cities. (Constraints (3.4) serve as the full set of Miller-Tucker-Zemlin subtour elimination constraints.) Constraint (3.8) represents the non-negativity requirement for each aircraft's start time for each aircraft. Constraint (3.9) restricts the sequencing variables to take the value of 0 or 1.

As imbedded component of the ASP model is a Traveling Salesman Problem with time-windows which renders the problem NP-Hard. A tighter formulation that yields improved lower bounds is designed in the sequel using the Reformulation-Linearization Technique (RLT) of Sherali and Adams (1990, 1994) and Sherali et al. (1998).

3.3.2 Valid Inequalities

As a preliminary, we first develop a tighter lower bound $LB(j)$ on the conditional start time of aircraft j's operation, given that it is last in the sequence.

Proposition 1. The maximum of the following three components can be used as a lower bound on the start time of aircraft j's operation, given that it is last in the sequence.

$$LB(j) = \max\{r_j, \max_{\substack{i \in J \\ i \neq j}}[r_i + \min_{\substack{k \in J \\ k \neq i}} p_{ik}],$$

$$\min_{\substack{i \in J \\ i \neq j}} [r_i + p_{0i} + \min_{\substack{k \in J \\ k \neq i,j}} p_{ik} + \min_{\substack{t \in J \\ t \neq i,j}} p_{tj} + \textit{sum of}(n\text{-}3)\ \textit{smallest from the array} \min_{\substack{n \in J \\ n \neq i,j,m}} p_{mn}\ \textit{for}\ m \in J,\ m \neq i,j\ ]\}$$

Proof. The first term in $LB(j)$, merely states that the start time of aircraft j should be greater than its ready-time (r_j). For the second term, since $r_i + \min_{k \in J} p_{ik}$ represents the earliest completion time of aircraft i regardless of the next aircraft, this term asserts that the last aircraft j should begin later than any other aircraft's earliest completion time. For the third term, if we assume that any particular aircraft $i \neq j$ is the first in the sequence, then the earliest start time of the last aircraft j is greater than or equal to the minimum of the total sum of processing times for all aircraft following i. This lower bound is given by the sum of the following events:

(i) the start time of the first aircraft i, $r_i + p_{0i}$,

(ii) the minimum separation time between the first two aircraft, $\min_{\substack{k \in J \\ k \neq i,j}} p_{ik}$,

(iii) the minimum separation time between the last two aircraft, $\min_{\substack{t \in J \\ t \neq i,j}} p_{tj}$,

(iv) the minimum of the remaining (n-3) separation times from the array of $\min_{\substack{n \in J \\ n \neq i,j,m}} p_{mn}$ for $m \in J$, $m \neq i,j$.

Taking the smallest of this sum (i)-(iv) over $i \in J$, $i \neq j$ yields a valid value for $LB(j)$. This completes the proof. □

A set of valid inequalities to replace constraints (3.4) can be generated as in (3.4.1) and (3.4.2) of Proposition 2 and 3 below, in order to achieve tighter lower bounds on the problem.

Proposition 2 (Valid Inequality I for Constraint (3.4)). For $i = 0, \forall j \in J$, the following inequality is valid and tighter than the corresponding constraint (3.4).

$$t_j \geq \max\{r_j, p_{0j}\} - (1-x_{0j})\max\{0, p_{0j}-r_j\} + x_{j0}[LB(j)-r_j] \quad \forall j \in J \qquad (3.4.1)$$

where, $LB(j) \geq r_j$ is a lower bound on the start time for the aircraft j, given that it is the last aircraft in the sequence (see Proposition 1).

Proof. When $x_{0j} = 1$, we have $x_{j0} = 0$ and then $t_j \geq \max\{r_j, p_{0j}\}$ is valid. When $x_{0j} = 0$ and $x_{j0} = 0$, regardless of the values of p_{0j} and r_j, $t_j \geq \max\{r_j, p_{0j}\} - \max\{0, p_{0j}-r_j\}$ is valid, since the resulting inequality is always $t_j \geq r_j$. When $x_{0j} = 0$ and $x_{j0} = 1$, aircraft j is last in the sequence. From above, the inequality (3.4.1) reduces to $t_j \geq r_j + [LB(j) - r_j]$ which imposes the

valid inequality $t_j \geq LB(j)$. Moreover, noting that for $i = 0$, (3.4) is $t_j \geq p_{0j} - (1- x_{0j})(p_{0j}- r_j) = r_j + x_{0j}(p_{0j}- r_j)$ while (3.4.1) implies via its first two terms that $t_j \geq r_j + x_{0j} \max\{0, p_{0j}- r_j\}$, we have that (3.4.1) dominates (3.4) in the continuous sense. This completes the proof. □

Proposition 3 (Valid Inequality II for Constraint (3.4)) For $i,j \in J, i \neq j$, the following inequality is valid and tighter than the corresponding constraint (3.4).

$$t_j \geq t_i + p_{ij}\text{-}(1\text{-}x_{ij})(d_i\text{-}r_j + p_{ij}) + \grave{e}\, x_{ji} \qquad \forall i,j \in J, i \neq j \qquad (3.4.2)$$

where, $\grave{e} = \max[0, d_i - r_j - \max\{p_{ji}, r_i - r_j\}]$.

Proof. When $x_{ij} = 1$, we have $x_{ji} = 0$ and then $t_j \geq t_i + p_{ij}$ is valid. When $x_{ij} = 0$ and $x_{ji} = 0$, then (3.4.2) is again valid since $(t_j\text{-}r_j) \geq 0 \geq (t_i\text{-}d_i)$. Finally to make (3.4.2) valid whenever $x_{ij} = 0$ and $x_{ji} = 1$, $q \geq 0$ should be chosen so that $t_i - t_j \leq d_i - r_j - \grave{e}$ is true under this condition. Note that when $x_{ji} = 1$, we have $t_i = \max\{t_j + p_{ji}, r_i\}$ and therefore $(t_i - t_j) = \max\{p_{ji}, r_i - t_j\} \leq \max\{p_{ji}, r_i - r_j\}$. Hence, for (3.4.2) to be valid, we can set $d_i - r_j - \grave{e} = \max\{p_{ji}, r_i - r_j\}$ and since $\boldsymbol{q} \geq 0$, we can compute $\grave{e} = \max[0, d_i - r_j - \max\{p_{ji}, r_i - r_j\}]$. Moreover, because of the additional nonnegative term $q \cdot x_{ji}$, (3.4.2) is tighter than (3.4) in the continuous sense. This completes the proof. □

3.3.3 Enhanced Model Representation via an Application of RLT

1) Reformulation Phase

Using $x_{ij}^2 = x_{ij}$, $x_{ij}x_{ik} = 0$ $\forall j \neq k$, $x_{ij}x_{kj} = 0$ $\forall i \neq k$, and $x_{ij}x_{ji} = 0$ $\forall i \in J_0, j \in J_0, k \in J_0$, we can construct additional sets of constraints via (R1)-(R5) stated below.

(R1) Multiply the assignment constraint (3.5) for each $i \in J$ by its corresponding t_i. Similarly, multiply the assignment constraint (3.6) for each $j \in J$ by its corresponding t_j. Note that constraints resulting from multiplying (3.2) and (3.3) by these assignment constraints (3.5) and (3.6) are then redundant.

$$\sum_{j \in J_0, j \neq i} t_i x_{ij} = t_i \qquad \forall i \in J \qquad (R1.1)$$

$$\sum_{i \in J_0, i \neq j} t_j x_{ij} = t_j \qquad \forall j \in J. \qquad \text{(R1.2)}$$

(R2) Multiply the inequality constraints (3.2) and (3.3) by the bound-factors $x_{ij} \geq 0$ and $x_{ji} \geq 0$ for $\forall i,j \in J$:

Multiplication by $x_{ij} \geq 0$:

$$(t_j - r_j) x_{ij} \geq 0 \qquad \forall i \in J, j \in J, i \neq j \qquad \text{(R2.1)}$$

$$(d_j - t_j) x_{ij} \geq 0 \qquad \forall i \in J, j \in J, i \neq j. \qquad \text{(R2.2)}$$

Multiplication by $x_{ji} \geq 0$:

$$(t_j - r_j) x_{ji} \geq 0 \qquad \forall i \in J, j \in J, i \neq j \qquad \text{(R2.3)}$$

$$(d_j - t_j) x_{ji} \geq 0 \qquad \forall i \in J, j \in J, i \neq j. \qquad \text{(R2.4)}$$

Note that we can tighten the constraints (R2.1) and (R2.4), by using conditional logic as follows, noting that the factors multiplied by the variables x_{ij} and x_{ji}, respectively, are relevant only when these variables take on a value of 1.

$$[t_j - \max\{r_j, r_i + p_{ij}\}] \cdot x_{ij} \geq 0 \qquad \forall i \in J, j \in J, i \neq j \qquad \text{(R2.1′)}$$

$$[\min\{d_j, d_i - p_{ji}\} - t_j] \cdot x_{ji} \geq 0 \qquad \forall i \in J, j \in J, i \neq j. \qquad \text{(R2.4′)}$$

If we consider similar products with x_{0j} and x_{j0}, we can tighten these constraints further.

$$(t_j - r_j) x_{0j} = 0 \qquad \forall j \in J \qquad \text{(R2.5)}$$

$$(t_j - LB(j)) x_{j0} \geq 0 \qquad \forall j \in J \qquad \text{(R2.6)}$$

$$(d_j - t_j) x_{j0} \geq 0 \qquad \forall j \in J. \qquad \text{(R2.7)}$$

Remark 1: In (R2.6), r_j has been validly replaced by any lower bound $LB(j)$ on the start time for aircraft j's operation, given that j is the last aircraft. For this lower bound, we can

use any reasonable conditional completion time which can be computed with relative ease. Also, note that (R2.5) is written as an equality since either $x_{0j} = 0$, or if $x_{0j} = 1$, then we can begin the first aircraft j's operation at $t_j = r_j$. Furthermore, because of this, the constraint $(d_j - t_j)\, x_{0j} \geq 0$ is redundant and has hence been omitted.

(R3) Multiply constraints (3.2), (3.3) by two-aircraft subtour elimination constraint, $(1 - x_{ij} - x_{ji}) \geq 0$.

$$(t_j - r_j)(1 - x_{ij} - x_{ji}) \geq 0 \qquad \forall\, i \in J, j \in J, i \neq j \qquad \text{(R3.1)}$$

$$(d_j - t_j)(1 - x_{ij} - x_{ji}) \geq 0 \qquad \forall\, i \in J, j \in J, i \neq j\,. \qquad \text{(R3.2)}$$

Also, we can write similar constraints for the 0-index, while tightening these further using logical tests. Noting that the following are relevant only under the condition $x_{0j} = x_{j0} = 0$ (else the left-hand-side is zero below), we can tighten the bounds r_j and d_j on t_j to derive the following valid product constraints.

$$(1\text{-}x_{0j}\text{-}x_{j0})[t_j - \max\{r_j, \min_{\substack{i \in J \\ i \neq j}} (r_i + p_{ij})\}] \geq 0 \qquad \forall j \in J \qquad \text{(R3.1′)}$$

$$(1\text{-}x_{0j}\text{-}x_{j0})[\min\{d_j, \max_{\substack{i \in J \\ i \neq j}} (d_i - p_{ji})\} - t_j] \geq 0 \qquad \forall j \in J\,. \qquad \text{(R3.2′)}$$

(R4) Multiply (3.4.1) by the bound factors $x_{0j} \geq 0$, $x_{j0} \geq 0$ and the two-aircraft subtour elimination constraint $(1 - x_{0j} - x_{j0}) \geq 0 \;\; \forall j \in J$.

Multiplication by $x_{0j} \geq 0$:

$$t_j x_{0j} - x_{0j} \max\{r_j, p_{0j}\} \geq 0 \qquad \forall j \in J\,. \qquad \text{(R4.1)}$$

Multiplication by $x_{j0} \geq 0$:

$$t_j x_{j0} - x_{j0} \max\{r_j, p_{0j}\} + x_{j0} \max\{0,\, p_{0j} - r_j\} - x_{j0}[LB(j) - r_j] \geq 0 \;\; \forall j \in J$$

i.e. $t_j x_{j0} - x_{j0} LB(j) \geq 0 \qquad \forall j \in J\,.$ (R4.2)

Multiplication by $(1\text{-}x_{0j} - x_{j0}) \geq 0$:

$$t_j (1\text{-}x_{0j}\text{-}x_{j0}) - r_j (1 - x_{0j}\text{-}x_{j0}) \geq 0 \qquad \forall j \in J\,. \qquad \text{(R4.3)}$$

(R5) Multiply (3.4.2) by the bound factors $x_{ij} \geq 0$, $x_{ji} \geq 0$ and the two-aircraft subtour elimination constraint $(1 - x_{ij} - x_{ji}) \geq 0$ for $\forall i,j \in J, i \neq j$.

Multiplication by $x_{ij} \geq 0$:

$$t_j x_{ij} - t_i x_{ij} - p_{ij} x_{ij} \geq 0 \qquad \forall i,j \in J, i \neq j. \qquad \text{(R5.1)}$$

Multiplication by $x_{ji} \geq 0$:

$$t_j x_{ji} - t_i x_{ji} + (d_i - r_j - \grave{e}) x_{ji} \geq 0 \qquad \forall i,j \in J, i \neq j. \qquad \text{(R5.2)}$$

where, $\grave{e} = \max[0, d_i - r_j - \max\{p_{ji}, r_i - r_j\}]$

Multiplication by the two-aircraft subtour elimination constraint $(1 - x_{ij} - x_{ji}) \geq 0$:

$$t_j (1 - x_{ij} - x_{ji}) - t_i (1 - x_{ij} - x_{ji}) \geq (r_j - d_i)(1 - x_{ij} - x_{ji}) \qquad \forall i,j \in J, i \neq j. \qquad \text{(R5.3)}$$

(R6) For computational convenience, we can rewrite the objective function and constraint (3.1) as (3.1') and (3.1″) given below.

Minimize z

subject to

$$t_i + \sum_{\substack{j \neq i \\ j \in J_0}} p_{ij} x_{ij} \leq z \qquad \forall\, i \in J_0 \qquad \text{(3.1′)}$$

$$z = \sum_{i \in J} (t_i + p_{i0}) x_{i0} \qquad \text{(3.1″)}$$

Let us now multiply (3.1′) by $x_{ik} \geq 0$ and $(1 - x_{ik}) \geq 0$, $\forall\, k \in J_0, k \neq i$. This gives the following.

$$z x_{ik} \geq t_i x_{ik} + p_{ik} x_{ik} \qquad \forall\, i \in J_0, k \in J_0, i \neq k \qquad \text{(R6.1)}$$

$$z \geq (t_i + \sum_{\substack{j \neq i \\ j \in J_0}} p_{ij} x_{ij}) + (z x_{ik} - t_i x_{ik} - p_{ik} x_{ik}) \qquad \forall\, i \in J_0, k \in J_0, i \neq k \qquad \text{(R6.2)}$$

Similarly, the multiplication of (3.1″) by $x_{k0} \geq 0$, $\forall\, k \in J$, yields

$$z x_{k0} = (t_k + p_{k0}) x_{k0} \qquad \forall\, k \in J. \qquad \text{(R6.3)}$$

In the RLT context, when zx_{ik} is substituted by a single (independent) variable, (R6.1) will likely hold as an equality and hence return (R6.2) to (3.1′). To make (R6.1)-(R6.3) more useful, we can also multiply the assignment constraints (3.5) and (3.6) by z.

Multiplication of z by the assignment factor (3.5) yields, using (R6.3),

$$z = (t_i + p_{i0})x_{i0} + \sum_{\substack{j \neq i \\ j \in J}} zx_{ij} \qquad \forall\, i \in J_0 . \qquad \text{(R6.4)}$$

Multiplication of z by (3.6) gives the following, using (R6.3).

$$z = \sum_{i \in J} (t_i + p_{i0})x_{i0} \qquad \text{for } j = 0, \text{ and} \qquad \text{(R6.5)}$$

$$z = \sum_{\substack{i \neq j \\ i \in J_0}} zx_{ij} \qquad \forall\, j \in J . \qquad \text{(R6.6)}$$

Note that constraint (R6.5) is the same constraint as (3.1″) and can therefore be deleted. However, (R6.6) can be retained. The resulting reformulation is stated below.

Minimize z

subject to

$$zx_{ik} \geq t_i x_{ik} + p_{ik} x_{ik} \qquad \forall\, i \in J_0, \forall\, k \in J_0,\ i \neq k \qquad \text{(R6.1)}$$

$$z \geq (t_i + \sum_{\substack{j \neq i \\ j \in J_0}} p_{ij} x_{ij}) + (zx_{ik} - t_i x_{ik} - p_{ik} x_{ik}) \qquad \forall\, i \in J_0, \forall\, k \in J_0,\ i \neq k \qquad \text{(R6.2)}$$

$$zx_{k0} = (t_k + p_{k0})x_{k0} \qquad \forall\, k \in J \qquad \text{(R6.3)}$$

$$z = (t_i + p_{i0})x_{i0} + \sum_{\substack{j \neq i \\ j \in J}} zx_{ij} \qquad \forall\, i \in J \qquad \text{(R6.4)}$$

$$z = \sum_{i \in J} (t_i + p_{i0})x_{i0} \qquad \text{(R6.5)}$$

$$z = \sum_{\substack{i \neq j \\ i \in J_0}} zx_{ij} \qquad \forall\, j \in J \qquad \text{(R6.6)}$$

$$(t_j - r_j)x_{0j} = 0 \qquad \forall\, j \in J \qquad \text{(R2.5)}$$

$$[t_j - \max\,(r_j, r_i + p_{ij})]x_{ij} \geq 0 \qquad \forall\, i \in J, j \in J, i \neq j \qquad \text{(R2.1′)}$$

$$(t_j - r_j)x_{ji} \geq 0 \qquad \forall\, i \in J, j \in J, i \neq j \qquad \text{(R2.3)}$$

$$(t_j - LB(j))x_{j0} \geq 0 \qquad \forall\, j \in J \qquad \text{(R2.6)}$$

$$(t_j - r_j)(1 - x_{ij} - x_{ji}) \geq 0 \qquad \forall\, i \in J, j \in J, i \neq j \qquad \text{(R3.1)}$$

$$(1\text{-}x_{0j}\text{-}x_{j0})[t_j - \max\,(r_j, \min_{\substack{i \in J \\ i \neq j}} (r_i + p_{ij}))] \geq 0 \qquad \forall\, j \in J \qquad \text{(R3.1′)}$$

$(d_j - t_j)x_{ij} \geq 0$ $\forall\, i \in J, j \in J, i \neq j$ (R2.2)

$(d_j - t_j)x_{j0} \geq 0$ $\forall\, j \in J$ (R2.7)

$[\min\ (d_j, d_i - p_{ji}) - t_j]x_{ji} \geq 0$ $\forall\, i \in J, j \in J, i \neq j$ (R2.4′)

$(d_j - t_j)(1 - x_{ij} - x_{ji}) \geq 0$ $\forall\, i \in J, j \in J, i \neq j$ (R3.2)

$(1-x_{0j}-x_{j0})[\min\ (d_j, \max_{\substack{i \in J \\ i \neq j}}(d_i - p_{ji})) - t_j] \geq 0$ $\forall\, j \in J$ (R3.2′)

$t_j x_{0j} - x_{0j} \max\{r_j, p_{0j}\} \geq 0,$ $\forall j \in J$ (R4.1)

$t_j x_{j0} - x_{j0} LB(j) \geq 0,$ $\forall j \in J$ (R4.2)

$t_j(1 - x_{0j} - x_{j0}) - r_j(1 - x_{0j} - x_{j0}) \geq 0,$ $\forall j \in J$ (R4.3)

$t_j x_{ji} - t_i x_{ji} + x_{ji}(d_i - r_j - \grave{e}) \geq 0$ $\forall i, j \in J, i \neq j$ (R5.2)

$t_j(1 - x_{ij} - x_{ji}) - t_i(1 - x_{ij} - x_{ji}) + (r_j - d_i)(x_{ij} + x_{ji}) \geq (r_j - d_i)$

$\forall i, j \in J, i \neq j$ (R5.3)

$\sum_{j \neq i} t_i x_{ij} = t_i$ $\forall\, i \in J$ (R1.1)

$\sum_{i \neq j} t_j x_{ij} = t_j$ $\forall\, j \in J$ (R1.2)

$\sum_{\substack{j \neq i \\ j \in J_0}} x_{ij} = 1$ $\forall\, i \in J_0$ (3.5)

$\sum_{\substack{i \neq j \\ i \in J_0}} x_{ij} = 1$ $\forall\, j \in J_0$ (3.6)

$x_{ij} + x_{ji} \leq 1$ $\forall\, i \in J_0, j \in J_0, j \neq i$ (3.7)

$t_0 = 0,\ t_i \geq 0$ $\forall\, j \in J$ (3.8)

$x_{ij} \in \{0,1\}$ $\forall\, i \in J_0, j \in J_0, j \neq i$ (3.9)

where, $LB(j)$ is given by Preposition 1, and where $\grave{e} = \max[0, d_i - r_j - \max\{p_{ji}, r_i - r_j\}]$ (see Proposition 3).

2) Linearization Phase

Let $t_i x_{ij} = u_{ij,}\ t_j x_{ij} = v_{ij,}\ z x_{ij} = z_{ij}$. The linearization of the reformulated constraints can be represented as follows (we state the identity of the corresponding constraint from the reformulation phase on the left of each of the following linearized restrictions.)

Minimize z

subject to

(R6.1) : $z_{ik} \geq u_{ik} + p_{ik} x_{ik}$ $\quad \forall\ i \in J,\ k \in J,\ i \neq k$ (L6.1)

(R6.2) : $z \geq (t_i + \sum_{\substack{j \neq i \\ j \in J_0}} p_{ij} x_{ij}) + (z_{ik} - u_{ik} - p_{ik} x_{ik})$ $\quad \forall\ i \in J,\ k \in J,\ i \neq k$ (L6.2)

(R6.3) : $z_{k0} = u_{k0} + p_{k0} x_{k0}$ $\quad \forall\ k \in J$ (L6.3)

(R6.4) : $z = u_{i0} + p_{i0} x_{i0} + \sum_{\substack{j \neq i \\ j \in J}} z_{ij}$ $\quad \forall\ i \in J$ (L6.4)

(R6.5) : $z = \sum_{i \in J} (u_{i0} + p_{i0} x_{i0})$ (L6.5)

(R6.6) : $z = \sum_{\substack{i \neq j \\ i \in J_0}} z_{ij}$ $\quad \forall\ j \in J$ (L6.6)

(R2.5) : $v_{0j} - r_j x_{0j} = 0$ $\quad \forall\ j \in J$ (L2.5)

(R2.1′) $v_{ij} - \max\ (r_j, r_i + p_{ij}) x_{ij} \geq 0$ $\quad \forall\ i \in J,\ j \in J,\ i \neq j$ (L2.1′)

(R2.3) : $u_{ji} - r_j x_{ji} \geq 0$ $\quad \forall\ i \in J,\ j \in J,\ i \neq j$ (L2.3)

(R2.6) : $u_{j0} - LB(j) x_{j0} \geq 0$ $\quad \forall\ j \in J$ (L2.6)

(R3.1) : $t_j - v_{ij} - u_{ji} - r_j (1 - x_{ij} - x_{ji}) \geq 0$ $\quad \forall\ i \in J,\ j \in J,\ i \neq j$ (L3.1)

(R3.1') : $t_j - v_{0j} - u_{j0} - \max\ \{r_j, \min_{\substack{i \in J \\ i \neq j}} (r_i + p_{ij})\} (1 - x_{0j} - x_{j0}) \geq 0$ $\quad \forall\ j \in J$ (L3.1′)

(R2.2) : $d_j x_{ij} - v_{ij} \geq 0$ $\quad \forall\ i \in J,\ j \in J,\ i \neq j$ (L2.2)

(R2.7) : $d_j x_{j0} - u_{j0} \geq 0$ $\quad \forall\ j \in J$ (L2.7)

(R2.4′) : $\min\ (d_j, d_i - p_{ji}) x_{ji} - u_{ji} \geq 0$ $\quad \forall\ i \in J,\ j \in J,\ i \neq j$ (L2.4′)

(R3.2) : $d_j (1 - x_{ij} - x_{ji}) - t_j + v_{ij} + u_{ji} \geq 0$ $\quad \forall\ i \in J,\ j \in J,\ i \neq j$ (L3.2)

(R3.2′) : $\min\ \{d_j, \max_{\substack{i \in J \\ i \neq j}} (d_i - p_{ji})\} (1 - x_{0j} - x_{j0}) - t_j + v_{0j} + u_{j0} \geq 0$ $\quad \forall\ j \in J$ (L3.2′)

(R4.1): $v_{0j} - x_{0j} \max\{r_j, p_{0j}\} \geq 0$ $\quad \forall\ j \in J$ (L4.1)

(R4.2): $t_j - v_{0j} - u_{j0} + r_j x_{0j} + r_j x_{j0} \geq r_j$ $\quad \forall\ j \in J$ (L4.2)

(R4.3): $u_{j0} - x_{j0} LB(j) \geq 0$ $\quad \forall j \in J$ (L4.3)

(R5.2) : $u_{ji} - v_{ji} - (d_i - r_j - q) x_{ji} \geq 0$ $\quad \forall\ i \in J,\ j \in J,\ i \neq j$ (L5.2)

(R5.3) : $t_j - v_{ij} - u_{ji} - t_i + u_{ij} + v_{ji} + (r_j - d_i) x_{ij} + (r_j - d_i) x_{ji} \geq (r_j - d_i)$ (L5.3)

$$\forall\, i \in J_0, j \in J, i \neq j$$

$$\text{(R1.1)}: \sum_{j \in J_0, j \neq i} u_{ij} = t_i \qquad \forall\, i \in J \qquad \text{(L1.1)}$$

$$\text{(R1.2)}: \sum_{i \in J_0, i \neq j} v_{ij} = t_j \qquad \forall\, j \in J \qquad \text{(L1.2)}$$

$$\sum_{\substack{j \neq i \\ j \in J_0}} x_{ij} = 1 \qquad \forall\, i \in J_0 \qquad (3.5)$$

$$\sum_{\substack{i \neq j \\ i \in J_0}} x_{ij} = 1 \qquad \forall\, j \in J_0 \qquad (3.6)$$

$$x_{ij} + x_{ji} \leq 1 \qquad \forall\, i \in J_0, j \in J_0, j \neq i \qquad (3.7)$$

$$t_0 = 0,\ t_i \geq 0 \qquad \forall\, j \in J \qquad (3.8)$$

$$x_{ij} \in \{0,1\} \qquad \forall\, i \in J_0, j \in J_0, j \neq i \qquad (3.9)$$

$$z_{ij} \geq 0,\ u_{ij} \geq 0,\ v_{ij} \geq 0 \qquad \forall\, i \in J, j \in J, i \neq j$$

where, $LB(j)$ is given by Preposition 1, and where $\grave{e} = \max[0, d_i - r_j - \max\{p_{ji}, r_i - r_j\}]$ (see Proposition 3).

3.3.4 Modifying the Formulation

3.3.4.1 Consideration of More Than Two Consecutive Aircraft

Up to now, our minimum separation constraint assumes that we only need to maintain an adequate separation between two consecutive aircraft. However, even though this constraint is satisfied, the resulting solution can violate the minimum separation rule between certain operations belonging to a common class in the context of mixed operations. Figure 3.2 illustrates this difficulty. In this case, the minimum separation between consecutive operations (i.e., arrival *i* and departure *k*, departure *k* and arrival *j*) are satisfied, but the required separation between the two arriving aircraft (i.e., arrival *i* and arrival *j*) may be violated.

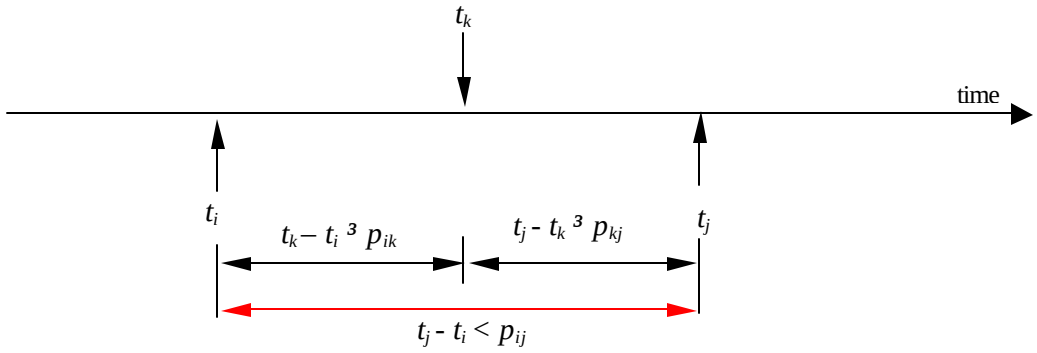


Figure 3.2 Checking Minimum Separation Constraints.

To protect against this event, we need another constraint set to enforce the minimum separation between the same types of operations as follows:

$$p_{ij}\hat{x}_{ij} - m_{ji}(1-\hat{x}_{ij}) \leq t_j - t_i \leq m_{ij}\hat{x}_{ij} - p_{ji}(1-\hat{x}_{ij}), \quad \forall i < j,\ (i,j) \in \textit{same class} \tag{3.10}$$

where,

$\hat{x}_{ij} = 1$ if $t_j > t_i$, and $\hat{x}_{ij} = 0$ if $t_i > t_j$

m_{ij}, m_{ji} : upper bounds on the separations between $i \rightarrow j$ and $j \rightarrow i$, respectively, $\forall i < j,\ (i,j) \in \textit{same class}$. We can define $m_{ij} = (d_j - r_i)$, and $m_{ji} = (d_i - r_j)$.

When $r_i + p_{ij} > d_j$, then it is clear that $\hat{x}_{ij} = 0$. (Note that in this case, we have $p_{ij} > m_{ij}$ and (3.10) leads to a contradiction when $\hat{x}_{ij} = 1$, hence also implying that $\hat{x}_{ij} = 0$.). In this case, we need to include the constraint $t_i \geq t_j + p_{ji}$. Similarly, if $r_j + p_{ji} > d_i$ (i.e., $p_{ji} > m_{ji}$), then we can fix $\hat{x}_{ij} = 1$ in (3.10) and include the relevant constraint $t_j \geq t_i + p_{ij}$. Additionally, in order to tighten the representation of this modified model, we can further relate the binary variable $\hat{x}_{ij}$ to the original model's variables $x_{ij}, x_{ji}, x_{ik}, x_{kj}$ via the valid inequalities given in Proposition 4 below.

Proposition 4. The following constraints are valid inequalities.

$$x_{ij} \leq \hat{x}_{ij} \leq 1 - x_{ji}, \qquad \forall\, i < j \text{ with } (i,j) \in \text{same class} \tag{3.11}$$

$$2x_{ij} + x_{ik} + x_{kj} - 1 \leq \hat{x}_{ij} \leq 2 - x_{jk} - x_{ki} - 2x_{ji}, \quad \forall\, i < j \text{ with } (i,j) \in \text{same class}, \forall k \neq i, j \tag{3.12.1}$$

$$x_{j0} \leq \hat{x}_{ij} \leq (1 - x_{i0}) \qquad \forall\, i < j \text{ with } (i,j) \in \text{same class} \tag{3.12.2}$$

$$x_{0i} \leq \hat{x}_{ij} \leq (1 - x_{0j}) \qquad \forall\, i < j \text{ with } (i,j) \in \text{same class}. \tag{3.12.3}$$

Proof. If $x_{ij} = 1$, which means $x_{ik} = x_{kj} = x_{ji} = 0$, and $(x_{ki}+x_{jk}) \leq 1$, (3.11) becomes $1 \leq \hat{x}_{ij} \leq 1$, and (3.12.1) becomes $1 \leq \hat{x}_{ij} \leq 2 - (x_{ki} + x_{jk})$. Hence, both (3.11) and (3.12.1) are valid in this case. If $x_{ij} = 0$, then equation (3.11) is clearly valid in both cases when $x_{ji} = 0$ (whence we get $0 \leq \hat{x}_{ij} \leq 1$), and $x_{ji} = 1$ (whence we get $\hat{x}_{ij} \equiv 0$). Furthermore, in this case (x_{ij}=0), (3.12.1) becomes

$$x_{ik} + x_{kj} - 1 \leq \hat{x}_{ij}, \text{ and} \tag{3.12.1.1}$$

$$\hat{x}_{ij} \leq 2 - x_{jk} - x_{ki} - 2x_{ji}. \tag{3.12.1.2}$$

Consider (3.12.1.1). If $x_{ik} = x_{kj} = 1$, then since $\hat{x}_{ij}$ must be 1, this is valid. Else, $x_{ik} + x_{kj} - 1 \leq 0$ and so (3.12.1.1) is implied by $\hat{x}_{ij} \geq 0$. Next, consider the inequality (3.12.1.2). If $x_{ji} = 1$, then $x_{jk} = x_{ki} = 0$ and since we must have $\hat{x}_{ij} = 0$, (3.12.1.2) is valid. On the other hand, suppose that $x_{ji} = 0$. Then, if $x_{jk} = x_{ki} = 1$, we must have $\hat{x}_{ij} = 0$, and therefore (3.12.1.2) is valid. Otherwise, we must have $2 - x_{jk} - x_{ki} \geq 1$ and so (3.12.1.2) is implied by $\hat{x}_{ij} \leq 1$. This validates (3.12.1).

For (3.12.2), note that $x_{i0} = 1$ and $x_{j0} = 1$, is impossible and so, the possible values of (x_{i0}, x_{j0}) are (0,0), (1,0) and (0,1). It is readily verified that each of these cases implies a valid set of bounds on $\hat{x}_{ij}$. Hence, (3.12.2) is valid. With the same argument, (3.12.3) is also valid for all possible cases. This completes the proof. □

Note that if the triangular inequality for the separation among aircraft triplets holds true, i.e., if $p_{ik} + p_{kj} \geq p_{ij} \quad \forall\, i < j \text{ with } (i, j) \in \textit{same class}, \forall k \neq i, j$, then we do not need to apply (3.10). As shown below, by successive applications of these triangular inequalities, if the separation between consecutive aircraft is enforced, then it holds between non-consecutive aircraft in the same class as well.

Proposition 5. If the triangular inequality is satisfied for all triplets of aircraft in J involving i in which i is first or last, or for all such triplets involving j, for any i and $j \in \textit{same class}$, then for any solution that satisfies the separation constraints for consecutive aircraft, we will also have that i and j satisfy their particular separation constraint.

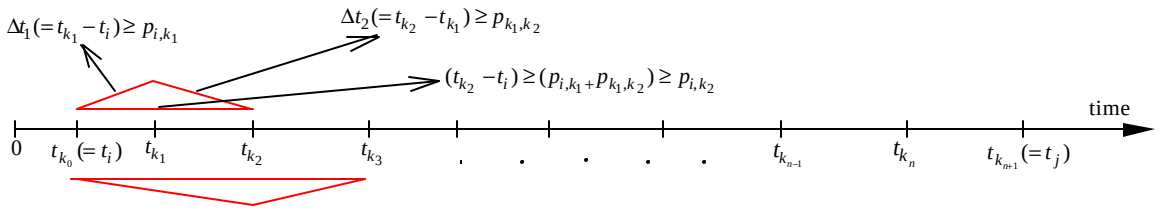


Figure 3.3 Triangular Inequality.

Proof. Without loss of generality, let $t_i < t_j$ (the case of $t_j < t_i$ is similar) and suppose that aircraft i and j are separated by some $n \geq 1$ aircraft $k_1, \ldots, k_n$. Then, we have by the feasibility of consecutive separation constraints that

$$(t_j - t_i) = \sum_{r=1}^{n+1} (t_{k_r} - t_{k_{r-1}}) \geq \sum_{r=1}^{n+1} p_{k_{r-1},k_r} \,, \qquad \text{(P5.1)}$$

where $t_{k_0} \equiv t_i$ and $t_{k_{n+1}} \equiv t_j$. Now, suppose that all triangular inequalities involving i are satisfied as stated in the Proposition (the case of j is similar). Then from (P5.1), we get

$$\begin{aligned}(t_j - t_i) &\geq p_{i,k_1} + p_{k_1,k_2} + p_{k_2,k_3} + \ldots + p_{k_{n-1},k_n} + p_{k_n,j} \\ &\geq p_{i,k_2} + p_{k_2,k_3} + \ldots + p_{k_{n-1},k_n} + p_{k_n,j} \\ &\geq p_{i,k_3} + \ldots + p_{k_{n-1},k_n} + p_{k_n,j} \geq \ldots\ldots \geq p_{i,j}\,.\end{aligned}$$

This completes the proof. □

To enforce separation between all pairs of aircraft in the same class, we need to impose the following constraints. Let V={i: some triangular inequalities are violated for a triplet involving i in which i is last or first}. The following is a summary of the constraint set generated to enforce aircraft separations.

$t_i \geq t_j + p_{ji}$ $\quad \forall\, i<j,\ (i,j) \in$ *same class* in V and $r_i + p_{ij} > d_j$

$t_j \geq t_i + p_{ij}$ $\quad \forall\, i<j,\ (i,j) \in$ *same class* in V and $r_j + p_{ji} > d_i$.

Also,

for all $i<j,\ (i,j) \in$ *same class* in V and $r_i + p_{ij} \leq d_j$, $r_j + p_{ji} \leq d_i$, (P5.2)

impose the following set of constraints, where, $m_{ij} = (d_j - r_i)$, $m_{ji} = (d_i - r_j)$

$$p_{ij}\hat{x}_{ij} - m_{ji}(1-\hat{x}_{ij}) \leq t_j - t_i \leq m_{ij}\hat{x}_{ij} - p_{ji}(1-\hat{x}_{ij})$$

$$x_{ij} \leq \hat{x}_{ij} \leq 1 - x_{ji}$$

$$2x_{ij} + x_{ik} + x_{kj} - 1 \leq \hat{x}_{ij} \leq 2 - x_{jk} - x_{ki} - 2x_{ji} \qquad \forall k \in J,\ k \neq i,j$$

$$x_{j0} \leq \hat{x}_{ij} \leq (1 - x_{i0})$$

$$x_{0i} \leq \hat{x}_{ij} \leq (1 - x_{0j})\ .$$

3.3.4.2 Applying RLT

Reformulation:

(R7) Multiply (3.10) by the bound factors $x_{ij} \geq 0$, $x_{ji} \geq 0$ and $(1 - x_{ij} - x_{ji}) \geq 0$ for $\forall i,j \in J, i \neq j$.

Multiplication by $x_{ij} \geq 0$:

$$(p_{ij} + m_{ji})x_{ij}\hat{x}_{ij} - m_{ji}x_{ij} + t_i x_{ij} - t_j x_{ij} \leq 0$$

$$t_j x_{ij} - t_i x_{ij} - (p_{ji} + m_{ij})x_{ij}\hat{x}_{ij} + p_{ji}x_{ij} \leq 0\ .$$

Multiplication by $x_{ji} \geq 0$:

$$(p_{ij} + m_{ji})x_{ji}\hat{x}_{ij} - m_{ji}x_{ji} - t_j x_{ji} + t_i x_{ji} \leq 0$$

$$t_j x_{ji} - t_i x_{ji} - (p_{ji} + m_{ij})x_{ji}\hat{x}_{ij} + p_{ji}x_{ji} \leq 0\ .$$

Multiplication by $(1-x_{ij}-x_{ji})\geq 0$:

$$(p_{ij}+m_{ji})\hat{x}_{ij}+t_i-t_j-[(p_{ij}+m_{ji})x_{ij}\hat{x}_{ij}-m_{ji}x_{ij}+t_ix_{ij}-t_jx_{ij}]$$
$$-[(p_{ij}+m_{ji})x_{ji}\hat{x}_{ij}-m_{ji}x_{ji}-t_jx_{ji}+t_ix_{ji}]\leq m_{ji}$$

$$t_j-t_i-(p_{ji}+m_{ij})\hat{x}_{ij}-[t_jx_{ij}-t_ix_{ij}-(p_{ji}+m_{ij})x_{ij}\hat{x}_{ij}+p_{ji}x_{ij}]$$
$$-[t_jx_{ji}-t_ix_{ji}-(p_{ji}+m_{ij})x_{ji}\hat{x}_{ij}+p_{ji}x_{ji}]\leq -p_{ji}.$$

Linearization:

Let $t_ix_{ij}=u_{ij}$ and $t_jx_{ij}=v_{ij}$ $\forall i,j$. Also note that $x_{ij}\hat{x}_{ij}\equiv x_{ij}$ and $x_{ji}\hat{x}_{ij}=0$.

$$p_{ij}x_{ij}\leq v_{ij}-u_{ij}\leq m_{ij}x_{ij} \tag{3.10.1}$$

$$-m_{ji}x_{ji}\leq u_{ji}-v_{ji}\leq -p_{ji}x_{ji} \tag{3.10.2}$$

$$p_{ij}(\hat{x}_{ij}-x_{ij})-m_{ji}(1-\hat{x}_{ij}-x_{ji})\leq (t_j-t_i)-(v_{ij}-u_{ij})-(u_{ji}-v_{ji})$$
$$\leq m_{ij}(\hat{x}_{ij}-x_{ij})-p_{ji}(1-\hat{x}_{ij}-x_{ji}). \tag{3.10.3}$$

Proposition 6. $Conv\{(t_i,t_j,\hat{x}_{ij}):(3.10)\ holds,\ \hat{x}_{ij}\ binary\}\equiv\{(3.10)\}\cap\{0\leq\hat{x}_{ij}\leq 1\}$ for each $(i,j)\in$ same class, $i<j$ such that $p_{ij}\leq m_{ij}$ and $p_{ji}\leq m_{ji}$ (i.e., for each (i,j) such that (P5.2) holds true).

Proof. Given any $(i,j)\in same\ class,\ i<j$, consider the following linear program for any c_i, c_j, c_{ij}.

$$\text{LP}: \max_{t_i,t_j,\hat{x}_{ij}} (c_it_i+c_jt_j+c_{ij}\hat{x}_{ij}) \tag{3.13.1}$$

subject to

$$p_{ij}\hat{x}_{ij}-m_{ji}(1-\hat{x}_{ij})\leq t_j-t_i\leq m_{ij}\hat{x}_{ij}-p_{ji}(1-\hat{x}_{ij}) \tag{3.13.2}$$

$$0\leq\hat{x}_{ij}\leq 1. \tag{3.13.3}$$

It is sufficient to show that if LP defined by (3.13) has an optimal solution then it has an optimum at which $\hat{x}_{ij}$=0 or 1. Denoting nonnegative slacks s_1 and s_2 in the two inequalities in (3.13.2), respectively, and eliminating t_j by substitution using the first equality, say, yields the following equivalent linear program

$\max\{(c_i+c_j)\,t_i+(c_{ij}+c_jp_{ij}+c_jm_{ji})\,\hat{x}_{ij}+c_js_1-c_jm_{ji}$:

$$\hat{x}_{ij}(p_{ij}+m_{ji}-p_{ji}-m_{ij})+s_1+s_2=(m_{ji}-p_{ji}),\ s_1\geq 0,\ s_2\geq 0,\ 0\leq\hat{x}_{ij}\leq 1\}.$$

Note that for an optimum to exist, we must have $(c_i + c_j) = 0$. Also, we then have that either $\hat{x}_{ij}$ is nonbasic at 0 or 1 at an optimal vertex, or if it is basic, then it must equal

$$\frac{(m_{ji} - p_{ji})}{(m_{ji} - p_{ji}) - (m_{ij} - p_{ij})} \geq 1.$$

Hence, in this case, we must have $(m_{ij}\text{-}p_{ij})=0$ and $\hat{x}_{ij}=1$ by feasibility. This completes the proof. □

Remark 2. Note that if we include the constraints $r_i \leq t_i \leq d_i$ and $r_j \leq t_j \leq d_j$ with (3.10), then the continuous relaxation can indeed have fractional extreme points. For example, we could have a vertex determined by the right-hand inequality in (3.13.2) binding along with $t_i = d_i$ and $t_j = d_j$ as three linearly independent constraints yielding the solution (where we have used $m_{ij} = d_j - r_i$)

$$t_i = d_i,\ t_j = d_j,\ \hat{x}_{ij} = \frac{d_j - d_i + p_{ji}}{d_j - r_i + p_{ji}}. \tag{3.14}$$

If $r_i < d_i < d_j$, this yields $0 < \hat{x}_{ij} < 1$. Also, this solution (3.14) is feasible to the left-hand inequality in (3.13.2) since this inequality is redundant whenever the right-hand inequality holds as an equality because of the hypothesis $p_{ij} \leq m_{ij}$ and $p_{ji} \leq m_{ji}$. Hence, this yields (3.14) as a fractional vertex.

Motivated by Remark 2, we now present a tightened representation of (3.10). Consider the representation of (3.10) given by the following *polynomial* set of constraints

$$(t_j - t_i)\hat{x}_{ij} \geq p_{ij}\hat{x}_{ij} \tag{3.15.1}$$

$$(t_i - t_j)(1 - \hat{x}_{ij}) \geq p_{ji}(1 - \hat{x}_{ij}) \tag{3.15.2}$$

$$r_i \leq t_i \leq d_i \tag{3.15.3}$$

$$r_j \leq t_j \leq d_j \tag{3.15.4}$$

$$\hat{x}_{ij} \text{ binary.} \tag{3.15.5}$$

By Sherali and Adams (1994), a polyhedral convex hull representation can be given by multiplying the inequalities in (3.15.1) - (3.15.4) by $\hat{x}_{ij}$, and $(1-\hat{x}_{ij})$, and substituting $\hat{x}_{ij}^2 = \hat{x}_{ij}$, $t_i \hat{x}_{ij} = r_{ij}$ and $t_j \hat{x}_{ij} = s_{ij}$. This gives (3.16) below as the convex hull of (3.15).

$$(s_{ij} - r_{ij}) \geq p_{ij}\hat{x}_{ij} \tag{3.16.1}$$

$$(t_i - t_j) - (r_{ij} - s_{ij}) \geq p_{ji}(1-\hat{x}_{ij}) \tag{3.16.2}$$

$$r_i \hat{x}_{ij} \leq r_{ij} \leq d_i \hat{x}_{ij},\ r_i(1-\hat{x}_{ij}) \leq (t_i - r_{ij}) \leq d_i(1-\hat{x}_{ij}) \tag{3.16.3}$$

$$r_j \hat{x}_{ij} \leq s_{ij} \leq d_j \hat{x}_{ij},\ r_j(1-\hat{x}_{ij}) \leq (t_j - s_{ij}) \leq d_j(1-\hat{x}_{ij}) \tag{3.16.4}$$

$$0 \leq \hat{x}_{ij} \leq 1. \tag{3.16.5}$$

Now, suppose that we further include within (3.15) the restrictions

$$x_{ij} \leq \hat{x}_{ij} \leq 1 - x_{ji}, \quad x_{ij}, x_{ji} \text{ binary.} \tag{3.11}$$

Note that by Sherali et al. (1996), the Special Structured RLT (SSRLT) factors are given by

$$S_1 = \{x_{ij} \geq 0, x_{ji} \geq 0, \hat{x}_{ij} - x_{ij} \geq 0, 1 - x_{ji} - \hat{x}_{ij} \geq 0\} \tag{3.17}$$

which collectively imply bounds of 0 and 1 on all the three binary variables as well as imply the 2-city DFJ subtour elimination constraint $x_{ij} + x_{ji} \leq 1$ (via the sum of the last two factors in S_1.) Using the fact that

$$x_{ij}\hat{x}_{ij} = x_{ij},\ x_{ji}\hat{x}_{ij} = 0, \text{ and } x_{ij}x_{ji} = 0, \tag{3.18}$$

the second-order factors obtained by pairwise (including self) products in (3.17) are given by S_1 itself, and hence, so are the third-order product factors. This means that we can generate *conv*{(3.15), (3.11)} by multiplying each of (3.15.1), (3.15.2), (3.15.3) and (3.15.4) by each factor defining S_1 in (3.17), including these factors (3.17) within the resulting constraint set, and substituting (3.18) along with

$$u_{ij} = t_i x_{ij},\ v_{ij} = t_j x_{ij},\ r_{ij} = t_i \hat{x}_{ij},\ s_{ij} = t_j \hat{x}_{ij},\ \hat{x}_{ij}^2 = \hat{x}_{ij}, x_{ij}^2 = x_{ij}, x_{ji}^2 = x_{ji}. \quad (3.19)$$

The resulting constraint sets are,

conv{(3.15), S_1}:

(3.15.1) $\cdot x_{ij}$: $v_{ij} - u_{ij} - p_{ij} x_{ij} \geq 0$ (3.15.1.11)

(3.15.1) $\cdot x_{ji}$: redundant.

(3.15.1) $\cdot (\hat{x}_{ij} - x_{ij})$: $(s_{ij} - r_{ij}) - (v_{ij} - u_{ij}) \geq p_{ij}(\hat{x}_{ij} - x_{ij})$ (3.15.1.12)

(3.15.1) $\cdot (1 - x_{ji} - \hat{x}_{ij})$: redundant.

(3.15.2) $\cdot x_{ij}$: redundant.

(3.15.2) $\cdot x_{ji}$: $v_{ji} - u_{ji} - p_{ji} x_{ji} \geq 0$. (3.15.2.11)

(3.15.2) $\cdot (\hat{x}_{ij} - x_{ij})$: redundant.

(3.15.2) $\cdot (1 - x_{ji} - \hat{x}_{ij})$: $(t_i - v_{ji} - r_{ij}) - (t_j - u_{ji} - s_{ij}) \geq p_{ji}(1 - x_{ji} - \hat{x}_{ij})$. (3.15.2.12)

(3.15.3) $\cdot x_{ij}$: $r_i x_{ij} \leq u_{ij} \leq d_i x_{ij}$. (3.15.3.11)

(3.15.3) $\cdot x_{ji}$: $r_i x_{ji} \leq v_{ji} \leq d_i x_{ji}$. (3.15.3.12)

(3.15.3) $\cdot (\hat{x}_{ij} - x_{ij})$: $r_i(\hat{x}_{ij} - x_{ij}) \leq r_{ij} - u_{ij} \leq d_i(\hat{x}_{ij} - x_{ij})$. (3.15.3.13)

(3.15.3) $\cdot (1 - x_{ji} - \hat{x}_{ij})$: $r_i(1 - x_{ji} - \hat{x}_{ij}) \leq t_i - v_{ji} - r_{ij} \leq d_i(1 - x_{ji} - \hat{x}_{ij})$. (3.15.3.14)

(3.15.4) $\cdot x_{ij}$: $r_j x_{ij} \leq v_{ij} \leq d_j x_{ij}$. (3.15.4.11)

(3.15.4) $\cdot x_{ji}$: $r_j x_{ji} \leq u_{ji} \leq d_j x_{ji}$. (3.15.4.12)

(3.15.4) $\cdot (\hat{x}_{ij} - x_{ij})$: $r_j(\hat{x}_{ij} - x_{ij}) \leq s_{ij} - v_{ij} \leq d_j(\hat{x}_{ij} - x_{ij})$. (3.15.4.13)

(3.15.4) $\cdot (1 - x_{ji} - \hat{x}_{ij})$: $r_j(1 - x_{ji} - \hat{x}_{ij}) \leq t_j - u_{ji} - s_{ij} \leq d_j(1 - x_{ji} - \hat{x}_{ij})$. (3.15.4.14)

For all $i < j$, $(i,j) \in$ *same class* and $i \in V$, $j \in V$, $r_i + p_{ij} \leq d_j$, $r_j + p_{ji} \leq d_i$, the above constraint set will be imposed.

We can also consider the convex hull representation for {(3.15), (3.12.2)}, {(3.15), (3.12.3)} using the special structured factors for (3.12.2) and (3.12.3) as follows:

$$S_2 = \{x_{i0} \geq 0,\ x_{j0} \geq 0,\ \hat{x}_{ij} - x_{j0} \geq 0,\ 1\text{-}\ x_{i0} - \hat{x}_{ij} \geq 0\}\,, \tag{3.20}$$

$$S_3 = \{x_{0i} \geq 0,\ x_{0j} \geq 0,\ \hat{x}_{ij} - x_{0i} \geq 0,\ 1\text{-}\ x_{0j} - \hat{x}_{ij} \geq 0\}\,. \tag{3.21}$$

Reformulation

Using $x_{j0}\hat{x}_{ij} = x_{j0}$, $x_{i0}\hat{x}_{ij} = 0$, $x_{0i}\hat{x}_{ij} = x_{0i}$, and $x_{0j}\hat{x}_{ij} = 0$, the reformulated constraint sets are,

conv{(3.15), S_2}:

(3.15.1) $\cdot\, x_{j0}$: $(t_j - t_i)x_{j0} \geq p_{ij}x_{j0}$

(3.15.1) $\cdot\, x_{i0}$: redundant.

(3.15.1) $\cdot\,(\hat{x}_{ij} - x_{j0})$: $(t_j - t_i)(\hat{x}_{ij} - x_{j0}) \geq p_{ij}(\hat{x}_{ij} - x_{j0})$

(3.15.1) $\cdot\,(1 - x_{i0} - \hat{x}_{ij})$: redundant.

(3.15.2) $\cdot\, x_{j0}$: redundant

(3.15.2) $\cdot\, x_{i0}$: $(t_i - t_j)x_{i0} \geq p_{ji}x_{i0}$.

(3.15.2) $\cdot\,(\hat{x}_{ij} - x_{j0})$: redundant

(3.15.2) $\cdot\,(1 - x_{i0} - \hat{x}_{ij})$: $(t_i - t_j)(1 - x_{i0} - \hat{x}_{ij}) \geq p_{ji}(1 - x_{i0} - \hat{x}_{ij})$.

(3.15.3) $\cdot\, x_{j0}$: $r_i x_{j0} \leq t_i x_{j0} \leq d_i x_{j0}$.

(3.15.3) $\cdot\, x_{i0}$: $r_i x_{i0} \leq t_i x_{i0} \leq d_i x_{i0}$.

(3.15.3) $\cdot\,(\hat{x}_{ij} - x_{j0})$: $r_i(\hat{x}_{ij} - x_{j0}) \leq t_i\hat{x}_{ij} - t_i x_{j0} \leq d_i(\hat{x}_{ij} - x_{j0})$.

(3.15.3) $\cdot\,(1 - x_{i0} - \hat{x}_{ij})$: $r_i(1 - x_{i0} - \hat{x}_{ij}) \leq t_i - t_i x_{i0} - t_i\hat{x}_{ij} \leq d_i(1 - x_{i0} - \hat{x}_{ij})$.

(3.15.4) $\cdot\, x_{j0}$: $r_j x_{j0} \leq t_j x_{j0} \leq d_j x_{j0}$.

(3.15.4) $\cdot\, x_{i0}$: $r_j x_{i0} \leq t_j x_{i0} \leq d_j x_{i0}$.

(3.15.4) $\cdot\,(\hat{x}_{ij} - x_{j0})$: $r_j(\hat{x}_{ij} - x_{j0}) \leq t_j\hat{x}_{ij} - t_j x_{j0} \leq d_j(\hat{x}_{ij} - x_{j0})$.

(3.15.4) $\cdot\,(1 - x_{i0} - \hat{x}_{ij})$: $r_j(1 - x_{i0} - \hat{x}_{ij}) \leq t_j - t_j x_{i0} - t_j\hat{x}_{ij} \leq d_j(1 - x_{i0} - \hat{x}_{ij})$.

conv{(3.15), S_3}:

(3.15.1) $\cdot\, x_{0i}$: $(t_j - t_i)x_{0i} \geq p_{ij}x_{0i}$

(3.15.1) $\cdot\, x_{0j}$: redundant.

(3.15.1) $\cdot\,(\hat{x}_{ij} - x_{0i})$: $(t_j - t_i)(\hat{x}_{ij} - x_{0i}) \geq p_{ij}(\hat{x}_{ij} - x_{0i})$

(3.15.1) $\cdot\,(1 - x_{0j} - \hat{x}_{ij})$: redundant.

(3.15.2) $\cdot\, x_{0i}$: redundant.

(3.15.2) $\cdot\, x_{0j}$: $(t_i - t_j)x_{0j} \geq p_{ji}x_{0j}$.

(3.15.2) $\cdot\,(\hat{x}_{ij} - x_{0i})$: redundant.

(3.15.2) $\cdot\,(1 - x_{0j} - \hat{x}_{ij})$: $(t_i - t_j)(1 - x_{0j} - \hat{x}_{ij}) \geq p_{ji}(1 - x_{0j} - \hat{x}_{ij})$.

(3.15.3) $\cdot x_{0i}$: $r_i x_{0i} \le t_i x_{0i} \le d_i x_{0i}$.

(3.15.3) $\cdot x_{0j}$: $r_i x_{0j} \le t_i x_{0j} \le d_i x_{0j}$.

(3.15.3) $\cdot (\hat{x}_{ij} - x_{0i})$: $r_i (\hat{x}_{ij} - x_{0i}) \le t_i \hat{x}_{ij} - t_i x_{0i} \le d_i (\hat{x}_{ij} - x_{0i})$.

(3.15.3) $\cdot (1 - x_{0j} - \hat{x}_{ij})$: $r_i (1 - x_{0j} - \hat{x}_{ij}) \le t_i - t_i x_{0j} - t_i \hat{x}_{ij} \le d_i (1 - x_{0j} - \hat{x}_{ij})$.

(3.15.4) $\cdot x_{0i}$: $r_j x_{0i} \le t_j x_{0i} \le d_j x_{0i}$.

(3.15.4) $\cdot x_{0j}$: $r_j x_{0j} \le t_j x_{0j} \le d_j x_{0j}$.

(3.15.4) $\cdot (\hat{x}_{ij} - x_{0i})$: $r_j (\hat{x}_{ij} - x_{0i}) \le t_j \hat{x}_{ij} - t_j x_{0i} \le d_j (\hat{x}_{ij} - x_{0i})$.

(3.15.4) $\cdot (1 - x_{0j} - \hat{x}_{ij})$: $r_j (1 - x_{0j} - \hat{x}_{ij}) \le t_j - t_j x_{0j} - t_j \hat{x}_{ij} \le d_j (1 - x_{0j} - \hat{x}_{ij})$.

Linearization

Using $r_{ij} = t_i \hat{x}_{ij}$, $s_{ij} = t_j \hat{x}_{ij}$, $u_{ij} = t_i x_{ij}$, $v_{ij} = t_j x_{ij}$, $f_{ij} = t_i x_{0j}$, $l_{ij} = t_i x_{j0}$, the linearized constraint sets are as follows:

conv{(3.15), S_2}:

(3.15.1) $\cdot x_{j0}$: $u_{j0} - l_{ij} \ge p_{ij} x_{j0}$ (3.15.1.21)

(3.15.1) $\cdot x_{i0}$: redundant.

(3.15.1) $\cdot (\hat{x}_{ij} - x_{j0})$: $(s_{ij} - r_{ij}) - (u_{j0} - l_{ij}) \ge p_{ij} (\hat{x}_{ij} - x_{j0})$ (3.15.1.22)

(3.15.1) $\cdot (1 - x_{i0} - \hat{x}_{ij})$: redundant.

(3.15.2) $\cdot x_{j0}$: redundant

(3.15.2) $\cdot x_{i0}$: $(u_{i0} - l_{ji}) \ge p_{ji} x_{i0}$. (3.15.2.21)

(3.15.2) $\cdot (\hat{x}_{ij} - x_{j0})$: redundant

(3.15.2) $\cdot (1 - x_{i0} - \hat{x}_{ij})$: $(t_i - u_{i0} - r_{ij}) - (t_j - l_{ji} - s_{ij}) \ge p_{ji} (1 - x_{i0} - \hat{x}_{ij})$. (3.15.2.22)

(3.15.3) $\cdot x_{j0}$: $r_i x_{j0} \le l_{ij} \le d_i x_{j0}$. (3.15.3.21)

(3.15.3) $\cdot x_{i0}$: $r_i x_{i0} \le u_{i0} \le d_i x_{i0}$. (3.15.3.22)

(3.15.3) $\cdot (\hat{x}_{ij} - x_{j0})$: $r_i (\hat{x}_{ij} - x_{j0}) \le r_{ij} - l_{ij} \le d_i (\hat{x}_{ij} - x_{j0})$. (3.15.3.23)

(3.15.3) $\cdot (1 - x_{i0} - \hat{x}_{ij})$: $r_i (1 - x_{i0} - \hat{x}_{ij}) \le t_i - u_{i0} - r_{ij} \le d_i (1 - x_{i0} - \hat{x}_{ij})$. (3.15.3.24)

(3.15.4) $\cdot x_{j0}$: $r_j x_{j0} \le u_{j0} \le d_j x_{j0}$. (3.15.4.21)

(3.15.4) $\cdot x_{i0}$: $r_j x_{i0} \le l_{ji} \le d_j x_{i0}$. (3.15.4.22)

(3.15.4) $\cdot (\hat{x}_{ij} - x_{j0})$: $r_j (\hat{x}_{ij} - x_{j0}) \le s_{ij} - u_{j0} \le d_j (\hat{x}_{ij} - x_{j0})$. (3.15.4.23)

(3.15.4) $\cdot (1 - x_{i0} - \hat{x}_{ij})$: $r_j (1 - x_{i0} - \hat{x}_{ij}) \le t_j - l_{ji} - s_{ij} \le d_j (1 - x_{i0} - \hat{x}_{ij})$. (3.15.4.24)

conv{(3.15), S_3}:

(3.15.1) $\cdot x_{0i}$: $f_{ji} - v_{0i} \geq p_{ij} x_{0i}$ (3.15.1.31)

(3.15.1) $\cdot x_{0j}$: redundant.

(3.15.1) $\cdot (\hat{x}_{ij} - x_{0i})$: $(s_{ij} - f_{ji}) - (r_{ij} - v_{0i}) \geq p_{ij}(\hat{x}_{ij} - x_{0i})$ (3.15.1.32)

(3.15.1) $\cdot (1 - x_{0j} - \hat{x}_{ij})$: redundant.

(3.15.2) $\cdot x_{0i}$: redundant.

(3.15.2) $\cdot x_{0j}$: $f_{ij} - v_{0j} \geq p_{ji} x_{0j}$. (3.15.2.31)

(3.15.2) $\cdot (\hat{x}_{ij} - x_{0i})$: redundant.

(3.15.2) $\cdot (1 - x_{0j} - \hat{x}_{ij})$: $(t_i - f_{ij} - r_{ij}) - (t_j - v_{0j} - s_{ij}) \geq p_{ji}(1 - x_{0j} - \hat{x}_{ij})$. (3.15.2.32)

(3.15.3) $\cdot x_{0i}$: $r_i x_{0i} \leq v_{0i} \leq d_i x_{0i}$. (3.15.3.31)

(3.15.3) $\cdot x_{0j}$: $r_i x_{0j} \leq f_{ij} \leq d_i x_{0j}$. (3.15.3.32)

(3.15.3) $\cdot (\hat{x}_{ij} - x_{0i})$: $r_i(\hat{x}_{ij} - x_{0i}) \leq r_{ij} - v_{0i} \leq d_i(\hat{x}_{ij} - x_{0i})$. (15.3-33)

(3.15.3) $\cdot (1 - x_{0j} - \hat{x}_{ij})$: $r_i(1 - x_{0j} - \hat{x}_{ij}) \leq t_i - f_{ij} - r_{ij} \leq d_i(1 - x_{0j} - \hat{x}_{ij})$. (3.15.3.34)

(3.15.4) $\cdot x_{0i}$: $r_j x_{0i} \leq f_{ji} \leq d_j x_{0i}$. (3.15.4.31)

(3.15.4) $\cdot x_{0j}$: $r_j x_{0j} \leq v_{0j} \leq d_j x_{0j}$. (3.15.4.32)

(3.15.4) $\cdot (\hat{x}_{ij} - x_{0i})$: $r_j(\hat{x}_{ij} - x_{0i}) \leq s_{ij} - f_{ji} \leq d_j(\hat{x}_{ij} - x_{0i})$. (3.15.4.33)

(3.15.4) $\cdot (1 - x_{0j} - \hat{x}_{ij})$: $r_j(1 - x_{0j} - \hat{x}_{ij}) \leq t_j - v_{0j} - s_{ij} \leq d_j(1 - x_{0j} - \hat{x}_{ij})$. (3.15.4.34)

3.3.4.3 Comparison of Constraints Sets

Let us defined constraint Set I as [(3.10), (3.11), (3.15.3), (3.15.4)]. Again, using $x_{ij}\hat{x}_{ij} = x_{ij}$, $x_{ji}\hat{x}_{ij} = 0$, and $x_{ij}x_{ji} = 0$, the application of SSRLT factor S_1 to constraint set I, SSRLT(I), yield the following.

SSRLT(I):

(3.10)·S_1:

$$(3.10)\cdot x_{ij}:\ p_{ij}x_{ij} \le v_{ij} - u_{ij} \le m_{ij}x_{ij} \tag{3.10.4}$$

$$(3.10)\cdot x_{ji}:\ -m_{ji}x_{ji} \le u_{ji} - v_{ji} \le -p_{ji}x_{ji} \tag{3.10.5}$$

$$(3.10)\cdot(\hat{x}_{ij} - x_{ij}):\ p_{ij}(\hat{x}_{ij} - x_{ij}) \le (s_{ij} - v_{ij}) - (r_{ji} - u_{ji}) \le m_{ij}(\hat{x}_{ij} - x_{ij}) \tag{3.10.6}$$

$$(3.10)\cdot(1 - x_{ji} - \hat{x}_{ij}):\ -m_{ji}(1 - \hat{x}_{ij} - x_{ji}) \le (t_j - t_i) - (s_{ij} - r_{ij}) - (u_{ji} - v_{ji}) \le -p_{ji}(1 - \hat{x}_{ij} - x_{ji}) \tag{3.10.7}$$

(3.11)·S_1: same to (3.11).

$(3.11)\cdot x_{ij}$: redundant.

$(3.11)\cdot x_{ji}$: redundant.

$(3.11)\cdot(\hat{x}_{ij} - x_{ij}):\ x_{ij} \le \hat{x}_{ij}$

$(3.11)\cdot(1 - x_{ji} - \hat{x}_{ij}):\ \hat{x}_{ij} \le 1 - x_{ji}$

(3.15.3) ·S_1: same to (3.15.3.1), (3.15.3.2), (3.15.3.3), (3.15.3.4).

$$(3.15.3)\cdot x_{ij}:\ r_i x_{ij} \le u_{ij} \le d_i x_{ij}. \tag{3.15.3.1}$$

$$(3.15.3)\cdot x_{ji}:\ r_i x_{ji} \le v_{ji} \le d_i x_{ji}. \tag{3.15.3.2}$$

$$(3.15.3)\cdot(\hat{x}_{ij} - x_{ij}):\ r_i(\hat{x}_{ij} - x_{ij}) \le r_{ij} - u_{ij} \le d_i(\hat{x}_{ij} - x_{ij}). \tag{3.15.3.3}$$

$$(3.15.3)\cdot(1 - x_{ji} - \hat{x}_{ij}):\ r_i(1 - x_{ji} - \hat{x}_{ij}) \le t_i - v_{ji} - r_{ij} \le d_i(1 - x_{ji} - \hat{x}_{ij}). \tag{3.15.3.4}$$

(3.15.4) ·S_1: same to (3.15.4.1), (3.15.4.2), (3.15.4.3), (3.15.4.4).

$$(3.15.4)\cdot x_{ij}:\ r_j x_{ij} \le v_{ij} \le d_j x_{ij}. \tag{3.15.4.1}$$

$$(3.15.4)\cdot x_{ji}:\ r_j x_{ji} \le u_{ji} \le d_j x_{ji}. \tag{3.15.4.2}$$

$$(3.15.4)\cdot(\hat{x}_{ij} - x_{ij}):\ r_j(\hat{x}_{ij} - x_{ij}) \le s_{ij} - v_{ij} \le d_j(\hat{x}_{ij} - x_{ij}). \tag{3.15.4.3}$$

$$(3.15.4)\cdot(1 - x_{ji} - \hat{x}_{ij}):\ r_j(1 - x_{ji} - \hat{x}_{ij}) \le t_j - u_{ji} - s_{ij} \le d_j(1 - x_{ji} - \hat{x}_{ij}). \tag{3.15.4.4}$$

Similarly, let us define Constraint Set II≡[(3.15.1), (3.15.2), (3.11), (3.15.3), (3.15.4)] and let us denote the application of SSRLT factor S_1 to this constraint as SSRLT(II).

Up to now, we have formulated two sets of constraints (i.e., Constraint Sets I and II) to enforce the separation rules for the same class operations. By applying RLT to Set I, and SSRLT to both Sets I and II, we have derived three sets of tightened constraints, RLT(I), SSRLT(I) and SSRLT(II) (see Table 3.6).

Table 3.6 Original and SSRLT Constraint Sets.

<table>
<tr><td rowspan="2">Original Constraints</td><td>Constraint Set (I): (3.10), (3.11), (3.15.3), (3.15.4)</td><td>Constraint Set (II):
(3.15.1), (3.15.2), (3.11), (3.15.3), (3.15.4)</td></tr>
<tr><td>$p_{ij}\hat{x}_{ij} - m_{ji}(1-\hat{x}_{ij}) \le t_j - t_i \le m_{ij}\hat{x}_{ij} - p_{ji}(1-\hat{x}_{ij})$ (3.10)

plus the following constraints:</td><td>$(t_j - t_i)\hat{x}_{ij} \ge p_{ij}\hat{x}_{ij}$ (3.15.1)
$(t_i - t_j)(1-\hat{x}_{ij}) \ge p_{ji}(1-\hat{x}_{ij})$ (3.15.2)
plus the following constraints:</td></tr>
<tr><td colspan="3">$x_{ij} \le \hat{x}_{ij} \le 1 - x_{ji}$ (3.11)
$r_i \le t_i \le d_i$ (3.15.3)
$r_j \le t_j \le d_j$ (3.15.4)</td></tr>
<tr><td rowspan="2">SSRLT Constraints by applying S_1</td><td>SSRLT(I)</td><td>SSRLT(II)</td></tr>
<tr><td>$p_{ij}x_{ij} \le v_{ij} - u_{ij} \le m_{ij}x_{ij}$ (3.10.4)
$-m_{ji}x_{ji} \le u_{ji} - v_{ji} \le -p_{ji}x_{ji}$ (3.10.5)
$p_{ij}(\hat{x}_{ij} - x_{ij}) \le (s_{ij} - v_{ij}) - (r_{ji} - u_{ji}) \le m_{ij}(\hat{x}_{ij} - x_{ij})$ (3.10.6)
$-m_{ji}(1-\hat{x}_{ij} - x_{ji}) \le (t_j - t_i) - (v_{ij} - u_{ij}) - (u_{ji} - v_{ji}) \le -p_{ji}(1-\hat{x}_{ij} - x_{ji})$ (3.10.7)

plus the following constraints:</td><td>$p_{ij}x_{ij} \le v_{ij} - u_{ij}$ (3.15.1.11)
$p_{ij}(\hat{x}_{ij} - x_{ij}) \le (s_{ij} - r_{ij}) - (v_{ij} - u_{ij})$ (3.15.1.12)

$u_{ji} - v_{ji} \le -p_{ji}x_{ji}$ (3.15.2.11)
$(t_j - t_i) - (s_{ij} - r_{ij}) - (u_{ji} - v_{ji}) \le -p_{ji}(1 - x_{ji} - \hat{x}_{ij})$ (3.15.2.12)
plus the following constraints:</td></tr>
<tr><td colspan="3">$x_{ij} \le \hat{x}_{ij} \le 1 - x_{ji}$ (3.11)
$r_i x_{ij} \le u_{ij} \le d_i x_{ij}$ (3.15.3.11)
$r_i x_{ji} \le v_{ji} \le d_i x_{ji}$ (3.15.3.12)
$r_i(\hat{x}_{ij} - x_{ij}) \le r_{ij} - u_{ij} \le d_i(\hat{x}_{ij} - x_{ij})$ (3.15.3.13)
$r_i(1 - x_{ji} - \hat{x}_{ij}) \le t_i - v_{ji} - r_{ij} \le d_i(1 - x_{ji} - \hat{x}_{ij})$ (3.15.3.14)

$r_j x_{ij} \le v_{ij} \le d_j x_{ij}$ (3.15.4.11)
$r_j x_{ji} \le u_{ji} \le d_j x_{ji}$ (3.15.4.12)
$r_j(\hat{x}_{ij} - x_{ij}) \le s_{ij} - v_{ij} \le d_j(\hat{x}_{ij} - x_{ij})$ (3.15.4.13)
$r_j(1 - x_{ji} - \hat{x}_{ij}) \le t_j - u_{ji} - s_{ij} \le d_j(1 - x_{ji} - \hat{x}_{ij})$ (3.15.4.14)</td></tr>
</table>

Let us now investigate the various implications among these constraint sets. As mentioned earlier, the RLT factor $1-x_{ij}-x_{ji}\geq 0$ is implied by summing the last two factors $\hat{x}_{ij}-x_{ij}\geq 0$ and $1-x_{ji}-\hat{x}_{ij}\geq 0$ defining S_1 in (3.18). Hence, since the other RLT factors $x_{ij}\geq 0$ and $x_{ji}\geq 0$ are also implied by S_1, we have that RLT(I) is implied by SSRLT(I).

For SSRLT(I) and SSRLT(II), the right-hand side (RHS) of (3.10.4) and (3.10.6) and the left-hand side (LHS) of (3.10.5) and (3.10.7) can be represented by (3.15.3.11)-(3.15.3.14) and (3.15.4.11)-(3.15.4.14) in the following way, noting that $m_{ij}=(d_j-r_i)$ and $m_{ji}=(d_i-r_j)$.

(3.15.4.11) – (3.15.3.11) = RHS of (3.10.4)
(3.15.4.12) – (3.15.3.12) = LHS of (3.10.5)
(3.15.4.13) – (3.15.3.13) = RHS of (3.10.6)
(3.15.4.14) – (3.15.3.14) = LHS of (3.10.7).

Consequently, all the inequalities having m_{ij} or m_{ji} in (3.10.4)-(3.10.7) can be eliminated. This reduces SSRLT(I) to SSRLT(II), thereby establishing their equivalence. Figure 3.4 presents the relationships among these various constraint sets. In practical terms, the aforementioned constraints can be generated by the following procedure.

For $i\in V$

For $j>i$, $j\in V$, $(i,j)\in$ same class

If $(r_i+p_{ij}>d_j)$, then $t_i\geq t_j+p_{ji}$

Else if $(r_j+p_{ji}>d_i)$, then $t_j\geq t_i+p_{ij}$

Else if $(r_i+p_{ij}\leq d_j$ and $r_j+p_{ji}\leq d_i)$, then

$$2x_{ij}+x_{ik}+x_{kj}-1\leq \hat{x}_{ij}\leq 2-x_{jk}-x_{ki}-2x_{ji} \qquad \forall k\in J,\ k\neq i,j$$

(3.15.1.11), (3.15.1.12), (3.15.2.11), (3.15.2.12),
(3.15.3.11), (3.15.3.12), (3.15.3.13), (3.15.3.14),
(3.15.4.11), (3.15.4.12), (3.15.4.13), (3.15.4.14).

(3.15.1.21), (3.15.1.22), (3.15.2.21), (3.15.2.22),
(3.15.3.21), (3.15.3.22), (3.15.3.23), (3.15.3.24),
(3.15.4.21), (3.15.4.22), (3.15.4.23), (3.15.4.24).

(3.15.1.31), (3.15.1.32), (3.15.2.31), (3.15.2.32),
(3.15.3.31), (3.15.3.32), (3.15.3.33), (3.15.3.34),
(3.15.4.31), (3.15.4.32), (3.15.4.33), (3.15.4.34).

Endif

End for *j*

End for *i*

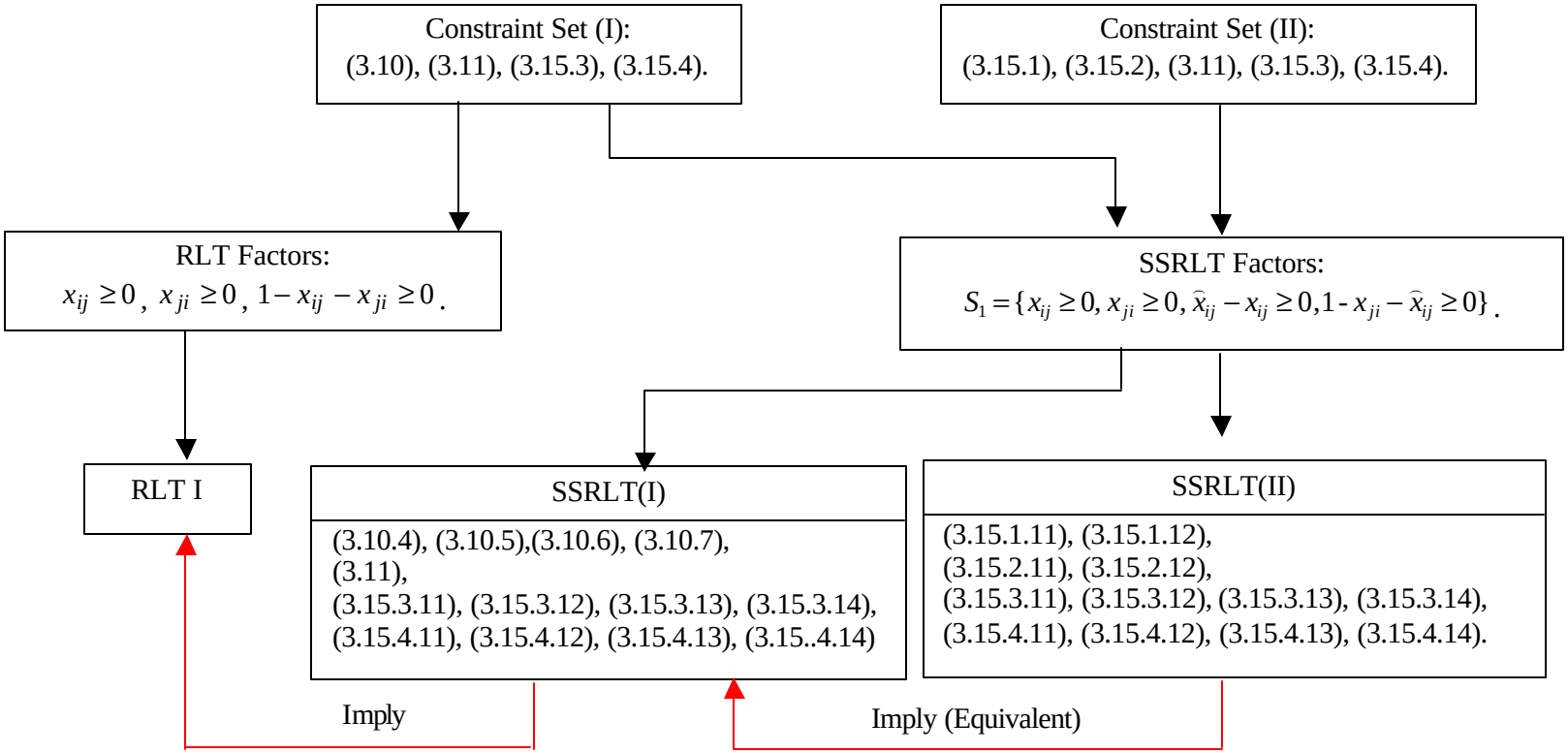


Figure 3.4 Relationships among the Constraint Sets.

.

3.3.4.4 Prefix and Probe Tests

Proposition 7. Let $I=\{1,\ldots,t\}$ be a collection of identical aircraft from the same class such that $r_1 \le r_2 \le \ldots \le r_t$ and $d_1 \le d_2 \le \ldots \le d_t$. Then we may fix $\hat{x}_{ij}=1 \ \forall\ i,j \in I, i<j$, and yet preserve optimality in ASP.

Proof. Consider any optimal solution to ASP and examine the sequence positions of aircraft from the set I. If these appear in the same order as in I, we are done. Otherwise, processing from the left in I, let the first out-of-sequence aircraft be in position p (possibly, $p=1$)and let the aircraft in this position be q. Note that $q>p$, and so, $r_p \le r_q$ and $d_p \le d_q$. Since $r_p \le r_q \le t_q < t_p \le d_p \le d_q$ by scheduling aircraft q at time t_p and aircraft p at time t_q, we would obtain a feasible solution having the same objective value. Continuing in this fashion, we would obtain an optimum with the aircraft in I sequenced in order $1,\ldots,t$. This completes the proof. □

Remark 3. Whenever we fix $\hat{x}_{ij}=1$,we automatically also have $x_{ji} \equiv 0$.

We can also perform probing tests on the x_{ij} variables based on the following concept. Suppose that we tentatively fix $x_{ij}=1$. If this results in an infeasible problem ASP, then we can permanently fix $x_{ij}=0$. Similarly, if fixing $x_{ij}=0$ renders the problem infeasible, then we can assert that we must have $x_{ij}=1$. To simplify the notation in this development, for each aircraft r, let

$$\Delta_r = \underset{s}{\text{minimum}} \{ p_{rs} \}$$

$$\text{and let } \hat{p}_{rs} = \begin{cases} p_{rs} \text{ if } s \text{ is in the same class as } r \\ \Delta_r \text{ otherwise} \end{cases} \qquad \forall \text{aircraft } s .$$

Proposition 8 (Probing on $x_{ij}=1$). Consider the pair of aircraft i and j associated with the (defined and not as yet fixed) variable x_{ij}. If either (a) $r_i+p_{ij}>d_j$, or (b) there exists an aircraft k for which the following two conditions hold:

$r_k + \hat{p}_{ki} > d_i$, or $r_k + \hat{p}_{ki} + p_{ij} > d_j$,

$\max\{r_i + p_{ij}, r_j\} + \hat{p}_{jk} > d_k$,

then we can optimally fix $x_{ij}=0$, assuming that an optimum exists.

Proof. Suppose that we consider any feasible solution in which $x_{ij}=1$. Clearly, if (a) holds true then we have a contradiction, and so we must have $x_{ij}=0$. Hence, suppose that (b) holds true. If the first part of condition (i) holds true, then k cannot precede i or else aircraft i cannot be feasibly scheduled, and similarly, if the second part of condition (i) holds true then again k cannot precede i or else aircraft j cannot be feasibly scheduled (given $x_{ij}=1$). On the other hand, condition (ii) asserts that aircraft k cannot be feasibly scheduled after i and j are consecutively scheduled. Together, these conditions imply that if $x_{ij}=1$, then there exists an aircraft k which cannot be feasibly scheduled, and so, we may fix $x_{ij}=0$. This completes the proof. □

Proposition 9 (Probing on $x_{ij}=0$). Consider the pair of aircraft i and j associated with the (defined and not as yet fixed) variable x_{ij}. If

$r_j + \hat{p}_{ji} > d_i$, and

$$\min_{\substack{k \neq i,j \\ r_i + p_{ik} \leq d_k}} [\max\{r_i + p_{ik}, r_k\} + \hat{p}_{kj}] > d_j,$$

then we may optimally fix $x_{ij}=1$, assuming that an optimum exists.

Proof. Suppose that we consider any feasible solution in which $x_{ij}=0$. Condition (i) asserts that j cannot precede i, and condition (ii) asserts that for any possible aircraft $k \neq i,j$ that immediately follows i, it would not be feasible to subsequently schedule aircraft j. Hence, aircraft j would remain unscheduled unless it immediately succeeds aircraft i, i.e., $x_{ij}=1$. This completes the proof. □

Remark 4. Naturally, if i and j belong to the same class and we fix $x_{ij}=1$ then we can also fix $\widehat{x}_{ij}=1$.

Figure 3.5 presents the scheme to prefix certain $\hat{x}_{ij}$ variables based on symmetry using the Proposition 7, 8 and 9.

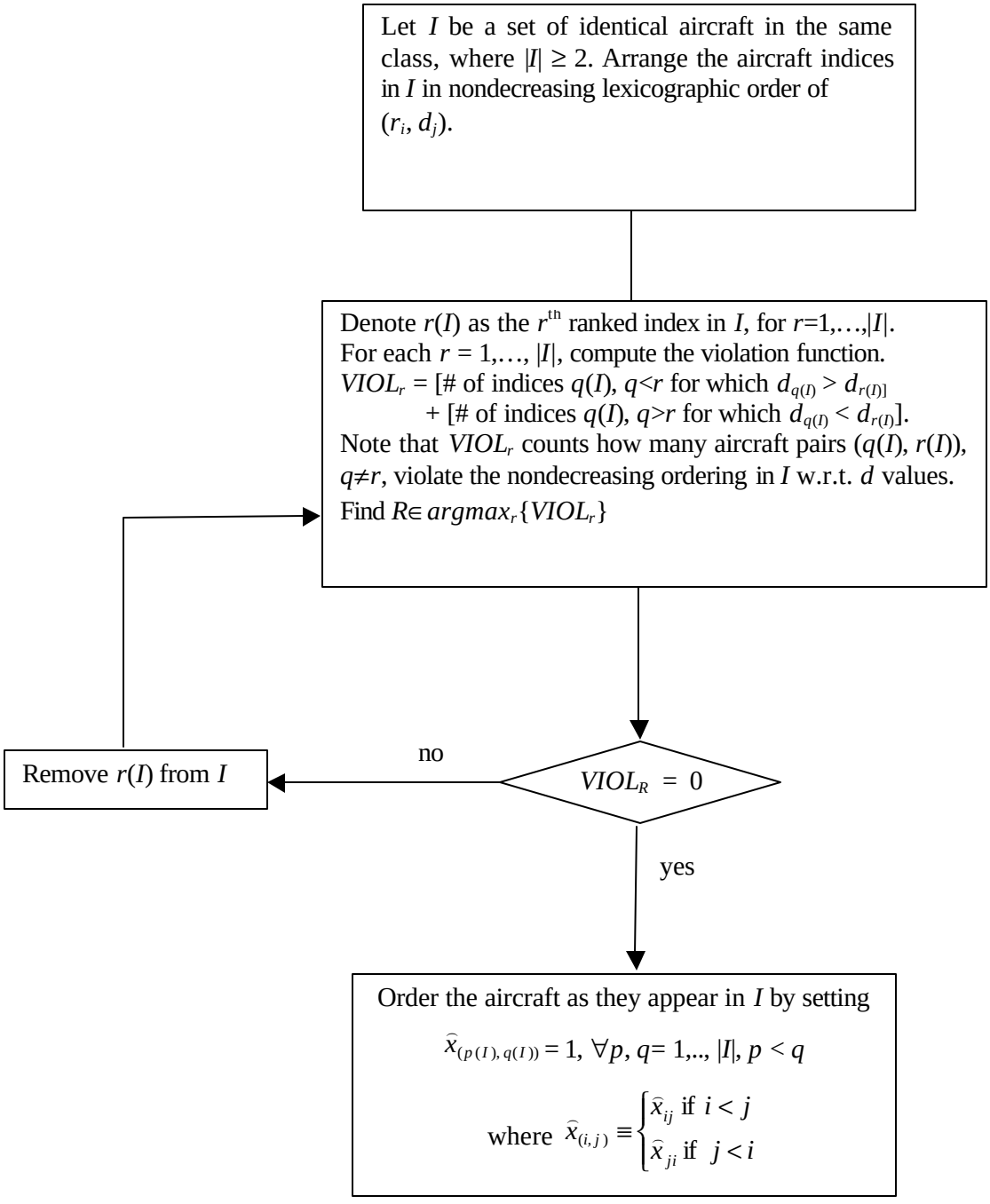


Figure 3.5 Prefixing Variables.

3.3.5 Computational Results

Figure 3.6 illustrates computational result of a simple case having 10 flights. As seen in the figure, the completion time of the first-come-first-served (FCFS) sequence and the optimal sequence are 624 seconds and 518 seconds, respectively. Hence by changing the traditional FCFS sequence intelligently, about 17% of the overall operational time can be saved.

We consider 12 problem instances having specified hypothetical flight schedules. The results given in Tables 3.7 and 3.8 indicate that the linear programming relaxation of the original traditional model seems to yield a significant gap from optimality that needs to be resolved. On the other hand, the proposed reformulated ASP problems yield tighter representations, producing significantly improved lower bounds. As a result, several previously unsolvable instances are now computationally tractable within the set limits. Nevertheless for real-time implementation, the CPU times (seconds) measured on an HP workstation indicate that we need to devise suitable heuristics as an alternative for deriving good quality solutions.

Various types of problems are defined as below according to the application of RLT and SSRLT. Problem **O** is an original problem without applying RLT. **RLT1** is a problem resulted from applying RLT to problem **O.** Problem **RLT2** involves additional constraints set generated by applying SSRLT factor S_1 to constraints set (3.15). In problem **RLT3,** SSRLT factor S_1, S_2 and S_3 are applied to constraints set (3.15). Among the problems, **O**, **RLT1** and **RLT2** are implemented in this research.

O: Minimize z

subject to

(3.1), (3.2), (3.3), (3.4), (3.5), (3.6), (3.7), (3.8), (3.9), (3.10), (3.11), (3.12),

$t_i \geq t_j + p_{ji}$ $\quad \forall\, i < j,\ (i,j) \in$ *same class* in V and $r_i + p_{ij} > d_j$

$t_j \geq t_i + p_{ij}$ $\quad \forall\, i < j,\ (i,j) \in$ *same class* in V and $r_j + p_{ji} > d_i$

$\hat{x}_{ij}$ binary.

RLT1: Minimize z

subject to

(L2.5), (L2.1′), (L2.3), (L2.6), (L3.1), (L3.1′), (L2.2), (L2.7), (L2.4′), (L3.2), (L3.2′), (L4.1), (L4.2), (L4.3), (L5.2), (L5.3), (L1.1), (L1.2), (3.5), (3.6), (3.7), (3.8), (3.9), (3.10), (3.11), (3.12),

$t_i \geq t_j + p_{ji}$ $\quad \forall\, i < j,\ (i,j) \in$ *same class* in V and $r_i + p_{ij} > d_j$

$t_j \geq t_i + p_{ij}$ $\quad \forall\, i < j,\ (i,j) \in$ *same class* in V and $r_j + p_{ji} > d_i$

$u_{ij} \geq 0,\ v_{ij} \geq 0$ $\quad \forall\, i \in J, j \in J, i \neq j$

$\widehat{x}_{ij}$ binary.

RLT2: Minimize *z*

subject to

(L2.5), (L2.1′), (L2.3), (L2.6), (L3.1), (L3.1′), (L2.2), (L2.7), (L2.4′), (L3.2), (L3.2′), (L4.1), (L4.2), (L4.3), (L5.2), (L5.3), (L1.1), (L1.2), (3.5), (3.6), (3.7), (3.8), (3.9), (3.10), (3.11), (3.10), (3.11), (3.12),

SSRLT[*conv*{(3.15), S_1}],

$t_i \geq t_j + p_{ji}$ $\quad \forall\, i < j,\ (i,j) \in$ *same class* in V and $r_i + p_{ij} > d_j$

$t_j \geq t_i + p_{ij}$ $\quad \forall\, i < j,\ (i,j) \in$ *same class* in V and $r_j + p_{ji} > d_i$

$u_{ij} \geq 0,\ v_{ij} \geq 0$ $\quad \forall\, i \in J, j \in J, i \neq j$

$r_{ij} \geq 0,\ s_{ij} \geq 0$ $\quad \forall\, i < j,\ (i,j) \in$ *same class*, $i \in V, j \in V$ and $r_i + p_{ij} \leq d_j$, $r_j + p_{ji} \leq d_i$

$\widehat{x}_{ij}$ binary.

RLT3: Minimize *z*

subject to

(L2.5), (L2.1′), (L2.3), (L2.6), (L3.1), (L3.1′), (L2.2), (L2.7), (L2.4′), (L3.2), (L3.2′), (L4.1), (L4.2), (L4.3), (L5.2), (L5.3), (L1.1), (L1.2), (3.5), (3.6), (3.7), (3.8), (3.9), (3.10), (3.11), (3.10), (3.11), (3.12),

SSRLT[*conv*{(3.15), S_1}],

SSRLT[*conv*{(3.15), S_2}],

SSRLT[*conv*{(3.15), S_3}],

$t_i \geq t_j + p_{ji}$ $\quad \forall\, i < j,\ (i,j) \in$ *same class* in V and $r_i + p_{ij} > d_j$

$t_j \geq t_i + p_{ij}$ $\quad \forall\, i < j,\ (i,j) \in$ *same class* in V and $r_j + p_{ji} > d_i$

$u_{ij} \geq 0,\ v_{ij} \geq 0$ $\quad \forall\, i \in J, j \in J, i \neq j$

$r_{ij} \geq 0,\ s_{ij} \geq 0,\ f_{ij} \geq 0,\ l_{ij} \geq 0,\ f_{ji} \geq 0,\ l_{ji} \geq 0$

$\forall\, i < j,\ (i,j) \in$ *same class*, $i \in V,\ j \in V$ and $r_i + p_{ij} \leq d_j$, $r_j + p_{ji} \leq d_i$

$\widehat{x}_{ij}$ binary.

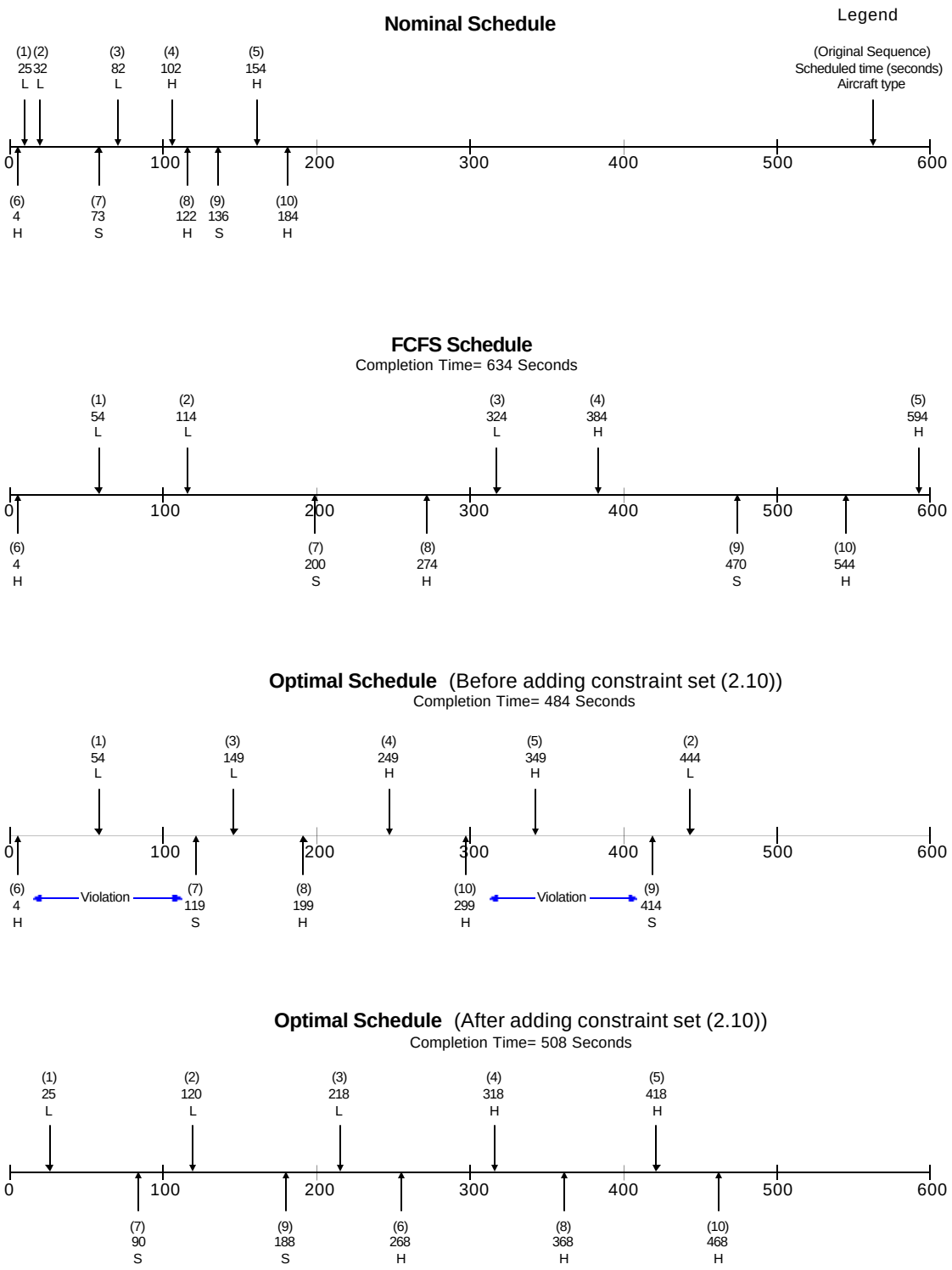


Figure 3.6 Illustrative Example.

Table 3.7 Computational Results (without using Prefixing Constraints).

Prob. Id	# acft	Data Intrvl (sec)	FCFS sol.	MIP value	O				RLT1			
					LP		MIP		LP		MIP	
					Value	time	nodes	Time	Value	Time	nodes	time
5_0	5	2000	1558	1558	1558.0	0.01	7	.02	1558	.12	0	.01
5_1	5	100	288	256	85.0	0.01	98	.11	230	.09	26	.23
5_2	5	100	263	234	79.0	0.02	223	.31	231	.12	34	.26
5_3	5	100	266	266	92.1	0.02	245	.31	238	.14	36	.40
5_4	5	100	260	260	84.9	0.01	199	.26	232	.11	59	.57
5_5	5	100	380	263	84.0	0.01	72	.09	262	.13	32	.35
10_0	10	2000	1883	1883	1871.0	0.03	46932	127.00	1873	3.30	47	32.14
10_3	10	600	750	686	500.0	0.06	27152	143.72	531	2.90	7806	784.51
10_4	10	600	626	492	305.0	0.03	300,000	1250.36	464	1.43	22147	2006.78
10_5	10	300	634	518	243.0	0.03	300,000	1275.90	479	2.42	22615	3646.54
10_6	10	200	715	524	185.0	0.03	300,000	*1283.90	489	1.12	45320	3612.37
15_0	15	2000	2095	2095	2046.0	0.04	300,000	1286.99	2077	19.67	1657	299.57
15_1	15	500	948									
15_3	15	500	967									

Prob. id	# acft	Data Intrvl (sec)	FCFS sol.	MIP value	RLT2							
					LP		MIP					
					Value	Time	nodes	Time				
5_0	5	2000	1558	1558	1558	.11	0	.01				
5_1	5	100	288	256	230	.12	32	.33				
5_2	5	100	263	234	231	.14	21	.18				
5_3	5	100	266	266	238	.15	42	.50				
5_4	5	100	260	260	232	.16	65	.66				
5_5	5	100	380	263	262	.21	24	.35				
10_0	10	2000	1883	1883	1883	3.39	0	.02				
10_3	10	600	750	686	531	5.49	2878	573.50				
10_4	10	600	626	492	464	2.21	20542	2878.87				
10_5	10	300	634	518	479	2.73	17413	2774.44				
10_6	10	200	715	524	489	2.15	48953	4791.63				
15_0	15	2000	2095	2095	2077	11.53	60	218.30				
15_1	15	500	948									
15_3	15	500	967									

*not optimal (due to the maximum nodes limit).

Table 3.8 Computational Results (using Prefixing Constraints).

Prob. Id	# acft	Data Intrvl (sec)	FCFS sol.	MIP value	O				RLT1			
					LP		MIP		LP		MIP	
					Value	time	nodes	Time	Value	Time	nodes	time
5_0	5	2000	1558	1558								
5_1	5	100	288	256								
5_2	5	100	263	234								
5_3	5	100	266	266								
5_4	5	100	260	260								
5_5	5	100	380	263								
10_0	10	2000	1883	1883	1871.6	0.02	3792	9.80	1873	2.51	50	29.57
10_3	10	600	750	686	551.0	0.04	6721	31.50	551	1.87	5299	1183.42
10_4	10	600	626	492	305.0	0.02	120518	442.98	464	1.22	2973	262.96
10_5	10	300	634	518	271.0	0.03	33793	127.77	479	1.23	4113	379.53
10_6	10	200	715	524	264.0	0.03	173046	660.00	489	1.22	8277	758.40
15_0	15	2000	2095									
15_1	15	500	948									
15_3	15	500	967									

Prob. id	# acft	Data Intrvl (sec)	FCFS sol.	MIP value	RLT2							
					LP		MIP					
					Value	Time	nodes	Time				
5_0	5	2000	1558	1558								
5_1	5	100	288	256								
5_2	5	100	263	234								
5_3	5	100	266	266								
5_4	5	100	260	260								
5_5	5	100	380	263								
10_0	10	2000	1883	1883	1883	1.90	14	18.22				
10_3	10	600	750	686	551	2.42	9358	3258.65				
10_4	10	600	626	492	464	1.48	3097	401.98				
10_5	10	300	634	518	479	1.40	3774	439.92				
10_6	10	200	715	524	489	1.38	9835	1263.82				
15_0	15	2000	2095									
15_1	15	500	948									
15_3	15	500	967									

3.4 Heuristics for the Aircraft Sequencing Problem (ASP)

Note that the aircraft sequencing decisions need to be made in real-time within a rolling horizon framework. As such, the computational time required to solve reasonably sized instances of ASP is of paramount importance. In such a context, both dynamic and integer programming approaches are not likely to be appropriate for implementation, especially when the problem size is relatively large. Hence, we propose the use of heuristic methods which provide solutions in a reasonable time. Although these solutions might be sub-optimal, they are motivated by partial or relaxed optimality considerations and are likely to be near-optimal.

3.4.1 Tour Building Process

3.4.1.1 FCFS Sequence

For the tour building procedure, instead of devising a new tour building procedure, the FCFS sequence (with landing priority) can be used as an initial tour. The motivation of this is that the FCFS sequence is the strategy currently applied in air traffic control, and we are seeking to improve the solution provided by it.

3.4.1.2 LP-based Heuristic

Having determined an LP-RLT solution $\overline{x}$ for an ASP, we can use this solution to construct an initial tour via a rounding process. (Among various types of RLT formulations shown in section 3.3.6, the RLT1 can be used for this purpose. This is because comparing with other RLTs, the computation time for LP of RLT1 is relatively small without losing the tightness of LP solution. See Table 3.7.) To do this, we develop two sequential rounding heuristics for the obtained LP solution.

Method 1: Round $\overline{x}$ based on FCFS or cost/delay considerations.

Method 2: Suppose that $\overline{x}$ is fractional. Solve $min \Sigma_i \Sigma_j \overline{x}_{ij} x_{ij}$, subject to assignment constraints, and hence obtain the solution $\hat{x}$. If $\hat{x}$ is a tour, then use this as the rounded tour. If $\hat{x}$ has subtours, construct a complete tour by using a FCFS-based strategy.

For Method 1, the following steps are performed:

Step 0 (*Selection of variables from the solution*): From the LP solution $\bar{x}$, for each i, find $j(i) \in \text{argmax}_j\{\bar{x}_{ij}\}$. Initialize the set of subtours as null.

Step 1 (*Sort and store the selected variables*): Sort the selected values $\bar{x}_{i,j(i)}\ \forall i$ in nonincreasing order, and store the corresponding ordered list of variables in an array $X(\cdot)$ of size N.

Step 2 (*Insert nodes in the current set of subtours*): Remove the first element $x_{i,j(i)}$ from $X(\cdot)$, and reduce the size of $X(\cdot)$ by one. Insert the nodes i and $j(i)$ in a current subtour by applying one of following rules.

Rule 1: If both i and $j(i)$ are currently included in the current set of subtours, go to Step 3.

Rule 2: If both i and $j(i)$ are not included in any current subtour, generate a new subtour by connecting nodes $0 \rightarrow i \rightarrow j(i) \rightarrow 0$.

Rule 3: If only i is included in some current subtour, $i \neq 0$, insert $j(i)$ between node i and the node following i in this subtour.

Rule 4: If only $j(i)$ is included in some current subtour, $j(i) \neq 0$, insert i between node $j(i)$ and its predecessor node in this subtour.

Rule 5: In case the selected variable is of the type x_{0j} (i.e., $i = 0$ with $j \equiv j(0)$ not included in any subtour), calculate the savings obtained by connecting j to node 0 in each of the current subtours. The savings are given by the following expression:

$$\text{saving}_{jf_m} = p_{0f_m} + p_{j0} - p_{jf_m},$$

where, f_m is the first node after node 0 in the m^{th} subtour.

Connect j as the immediate successor to node 0 in the subtour which yields the largest savings.

Rule 6: In case the selected variable is of the type x_{i0} (i.e., $j(i)=0$), with i not included in any subtour, calculate the savings obtained by connecting i to node 0 in each of the current subtours. The savings are given by the following expression:

$$\text{saving}_{l_m i} = p_{l_m 0} + p_{0i} - p_{l_m i},$$

where, l_m is the last node before node 0 in the m^{th} subtour.

Connect i as the immediate predecessor of node 0 in the subtour which yields the largest savings.

Step 3 (*Repeat*): If the array $X(\cdot)$ is not empty, go to Step 2. Else, go to Step 4.

Step 4 (*Connect subtours*): If the current solution is a complete tour, stop. Else (i.e., the current solution has more than one subtour), compute all savings by connecting the last node in a subtour to the first node in another for all pairs of subtours. The savings for connecting the last node l_m in the m^{th} subtour and the first node f_n in the n^{th} subtour is given by the following expression.

$$\text{saving}_{mn} = p_{l_m 0} + p_{0 f_n} - p_{l_m f_n}, \qquad \forall\, m, n \in \{\text{the index set for subtours}\},\ m \neq n.$$

Join the pair of subtours that yields the highest savings into a single subtour according to the corresponding connection. Repeat Step 4 until a complete tour is obtained.

Table 3.9 shows an example of this initial tour building procedure for the case having 10 aircraft. The first two columns in the table present the sorted LP-solution values, and the rules applied to insert the nodes are shown in the third column.

Table 3.9 An Example of Heuristic Method for ASP.

Variable	LP-solution	Rule for insertion
$x_{3,10}$	0.93	rule 2
$x_{7,3}$	0.91	rule 4
$x_{4,9}$	0.84	rule 2
$x_{9,5}$	0.81	rule 3
$x_{8,4}$	0.78	rule 4
$x_{0,6}$	0.70	rule 5
$x_{1,0}$	0.66	rule 6
$x_{5,8}$	0.57	rule 1
$x_{6,2}$	0.54	rule 3
$x_{10,2}$	0.45	rule 1

Figure 3.7 depicts the stepwise process for this example. For instance, Figure 3.7(6) illustrates two choices of connecting flight 6 to node 0, and the best resulting sequence after applying Rule 4 is shown in Figure 3.7(7). Figures 3.7(11) and 3.7(12) illustrate two possible ways to connect the current pair of subtours, and Figure 3.7(13) depicts the final selected sequence after joining these subtours.

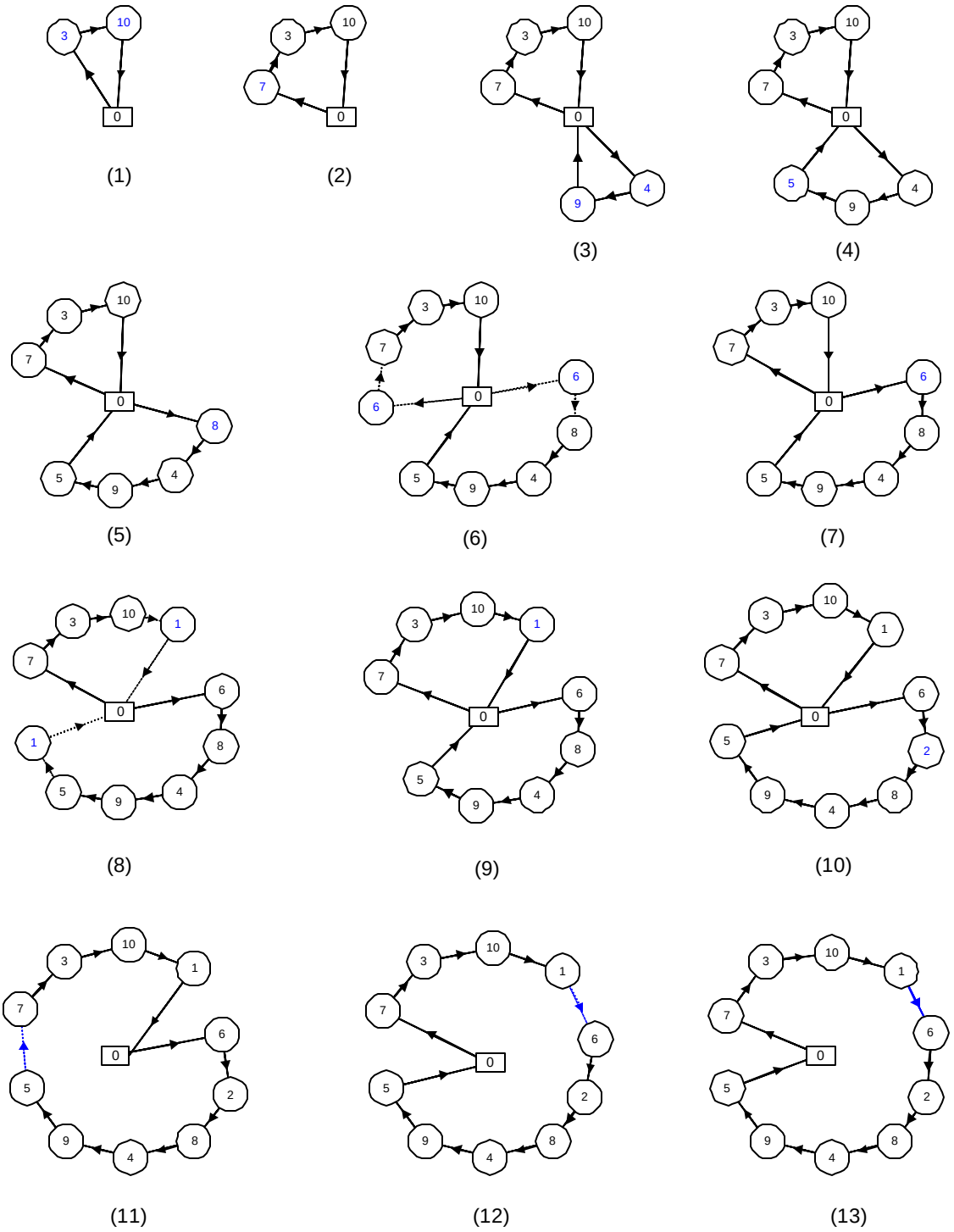


Figure 3.7 LP-based tour building procedure for ASP.

3.4.2 Improvement Heuristic

3.4.2.1 2-exchange Heuristic

The 2-exchange scheme developed by Lin (1965) can be applied as an improvement routine for ASP. Associated with a proposed exchange, the new schedule can be tested against the minimum separation rules by considering each affected flight and its three previous flights. This is necessary because the triangle inequality does not necessarily hold for the given edge weights. The data for the minimum separation rule has a largest separation of 196 seconds, which is required when a small aircraft follows a heavy one in an arrival sequence. In this case, two departing operations can successively intervene between these two arrivals without delaying (or pushing forward) the small aircraft schedule. Even though the separations between the small aircraft arrival and the two departures are adequate, there is still a chance that the separation rule between the heavy aircraft arrival and the small aircraft arrival is violated. Hence the foregoing three-previous-flight rule.

Given that a 2-exchange involves the substitution of two edges, $(i, i+1)$ and $(j, j+1)$ with two other edges (i, j) and $(i+1, j+1)$ as shown previously in Figure 2.5, the profitability of a proposed sequence is checked at the $(j+1)^{th}$ aircraft using the following condition:

> If the start time of the $(j+1)^{st}$ aircraft's operation in the suggested 2-exchange sequence is earlier than that in the current sequence, the new sequence is profitable.

In fact, this condition is not sufficient but is only a necessary condition for the improvement of the current sequence. The reason for this is that there still exists a possibility for some flight following the $(j+1)^{st}$ flight to be further delayed because of the separation constraints with respect to the revised two predecessors of the $(j+1)^{st}$ flight. This relaxed condition sometimes results in a non-monotone decreasing sequence of completion times. However, from computational experience, it is observed that in some cases, this relaxed condition is very helpful to attain improved solutions. The entire improvement process is repeated from the beginning whenever a profitable sequence is obtained, and is continued until no feasible and profitable sequence is found. The pseudo-code for this heuristic is shown in Figure 3.8.

```
iter = 0;
heurSeq = initial sequence from tour building;
prepareVP();
reduceTimeWindow();

do{
      findBetterTwoExchange = false;
      findBetterTwoExchange = heuristicASP ();
      iter++;
}while(findBetterTwoExchange is not found);


heuristicASP()
{
      int i, j, k;
      findBetterTwoExchange = false;
      for(i = 0 to totFlights –1){
          for(j = i+2 to totFlights){
              initialize tempSeq;
              for (k = 0 to tot + 1){
                   if(k <= i) tempSeq[k] = heurSeq[k];
                   else if (k > i  and k <= j){  // reversed sequence
                          tempSeq[k] = heurSeq[j - (k - i) + 1];
                          update tempSeq;

                          if(tempSeq is infeasible){
                             initialize tempSeq;
                             break;
                                 }
                        }
                   else if (k == j + 1){  // check feasibility and profitability
                          tempSeq[k] = heurSeq[k];
                          update tempSeq;
                          if(tempSeq is infeasible or not profitable){
                             initialize tempSeq;
                             break;
                          }
                   }
                   else if(k < totFlights + 1){
                          tempSeq[k] = heurSeq[k];
                          update tempSeq;
                          if(tempSeq is infeasible){
                             initialize tempSeq;
                             break;
                         }
                   }
                   else{ // i.e., k = tot + 1;
                          tempSeq[k] = heurSeq[k];
                          update tempSeq;
                          if(tempSeq is infeasible){
                              initialize tempSeq;
                             break;
                        }
                         else{// i.e., tempSeq is feasible
                            for (m = 0 to tot + 1)  heurSeq[m] = tempSeq[m]; // update heurSeq
                                     return findBetterTwoExchange = true;
                         }
                   }//end if

                }//for k
             }//for j
          }//for i
          return  findBetterTwoExchange = false;
}

update tempSeq{
// decide and update the current flight's start time considering the minimum separation rules between
//  this flight and three previous flights.
}
checkFeasiblity of tempSeq(){
// if the tempSeq’s start time is later than due-time then the tempSeq is not feasible.
}

checkProfitablity of tempSeq(curr, tSeq, hSeq){
 // if the tempSeq’s start time is less than the heurSeq’s start time, then the tempSeq is profiable.
}
```

Figure 3.8 Pseudo-code for the ASP heuristic.

3.4.2.2 2-swap Heuristic

Besides the 2-exchange procedure, we can apply the 2-node swapping method which swaps a node with one of its local neighbors that reside within some n nodes adjacent to the current node. Let us consider swapping an i^{th} with a j^{th} flight for $j>i$. The 2-swap heuristic would check feasibility for all the flights after the j^{th} flight, and would also check the profitability of this exchange at the $i+1^{st}$ flight. If the new starting time for the $i+1^{st}$ flight is earlier than the previous starting time, this swapped sequence is considered as profitable.

3.4.2.3 Enhancing Improvement Heuristics

For expediting the improvement procedure, two additional methods may be applied: 1) Exploiting a precedence relationship, and 2) reduction of time-windows.

Method 1 (*Exploiting precedence relationship*): In ASP, the precedence relations among the flights can be found by applying the following rules:

Rule 1: The sequence is symmetrically identical by feasibly interchanging the same types of operations involving the same aircraft type. As such, the FCFS rule could be applied to such flights. In other words, restrict $\text{flight}_i \prec \text{flight}_j$, if $\text{operation}_i = \text{operation}_j$, $(\text{aircraft type})_i = (\text{aircraft type})_j$, and $e_i \le e_j$.

Rule 2: Let $[e_i, l_i]$ and $[e_j, l_j]$ denote time-windows for flights i and j, respectively. If $l_i \le e_j$, then flight i should precede flight j in the final sequence.

Adopting Solomon's (1988) approach, the precedence information among the flights are stored in a matrix VP(i, j), where VP(i, j) is defined as follows.

$$\text{VP}(i,j) = \begin{cases} +1 \text{ if flight } i \text{ must precede flight } j, \\ 0 \text{ if no precedence relationship exists,} \\ -1 \text{ if flight } j \text{ must precede flight } i, \end{cases}$$

where, i and j are indices for flight identification.

Once VP(i, j) is available, a node precedence value, NP(i), for the i^{th} flight in the current sequence, is defined as follows:

NP(i) = the smallest number k, $k > i$+1, such that VP(j, k) = +1, for some $j \geq i$+1. If no such k exists, then NP(i) = N+1,
where, i is a position for the i^{th} flight in the current sequence, and N is the number of flights.

The NP(i) array can be obtained from VP(i, j) in $O(N^2)$ time by applying the following procedure, where N is the number of flights in the current sequence.

```
current_i = 1;
for(k = 2 to k < N + 1){
    for(j = current_i to j < k){
        NP(j – 1) = k;
        j_id = id of currSeq(j);
        k_id = id of currSeq(k);
        if(VP(j_id - 1, k_id - 1) = 1){
            current_i = j + 1;
            break;
                                                    }
    }//end for
}//end for
NP(N) = N+1;
NP(N+1) = N+1;
```

Henceforth, the improvement procedure can be expedited by adopting the following necessary condition for the feasibility of a 2-exchange as suggested by Psaraftis (1983b):

A necessary condition for the feasibility of a 2-exchange of arcs (i, i+1) and (j, j+1) with (i, j) and (i+1, j+1) is that $j <$ NP(i).

Let us consider an example case having 10 flights with the following current sequence.

Sequence	0	1	2	3	4	5	6	7	8	9	10	11
Flight Id.	f_0	f_6	f_7	f_8	F_1	f_2	f_9	f_{10}	f_3	f_4	f_5	f_0

Assuming that the precedence relations among the flights are $f_1 \prec f_4$, $f_1 \prec f_5$, $f_2 \prec f_3$, $f_4 \prec f_5$, $f_6 \prec f_7$, $f_6 \prec f_8$, and $f_7 \prec f_8$, the resulting NP(i) values for the current sequence are given as follows:

i	NP(i)
0	2
1	2
2	3
3	8
4	8
5	10
6	10
7	10
8	10
9	11
10	11

Method 2 (*Reduction of time-windows*): Exploiting the precedence relations imposed in the previous step, certain time-windows can be reduced as follows: Let $[e_i, l_i]$ and $[e_j, l_j]$ be time-windows for flights i and j, respectively. If flight i precedes flight j, then their time-windows can be reduced by letting $l_i = \min\{l_i, l_j - t_{ij}\}$ and $e_j = \max\{e_j, e_i + t_{ij}\}$.

The overall steps included in Methods 1 and 2 are depicted in Figure 3.9.

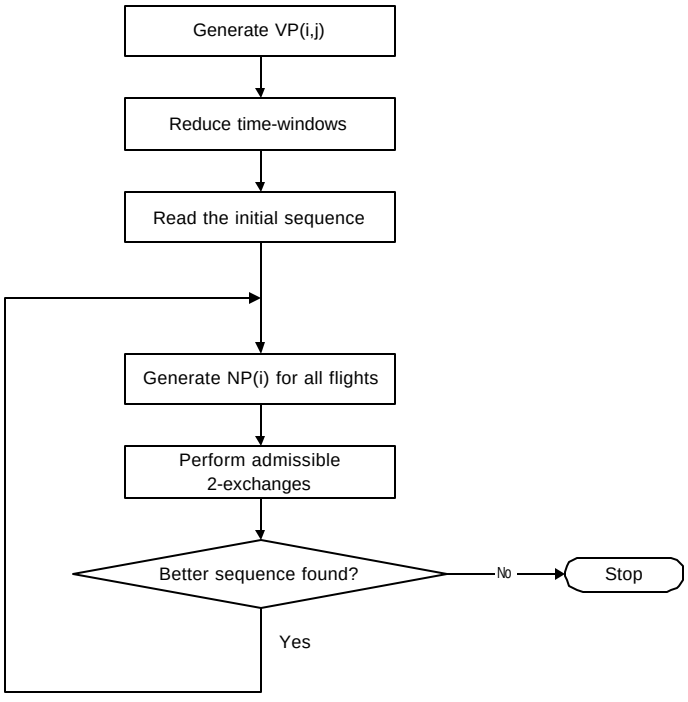


Figure 3.9 Flowchart of Improvement Procedure for ASP.

3.4.3 Computational Results

To test suggested heuristic methods, we performed computations using a set of randomly generated problems. For the tour building process, the FCFS sequences are used. The results obtained from 2-exchange improvement procedure are summarized in Table 3.10. In the problems 15_0 and 50_7, the time-windows for flights appearing later in the schedule dominate the overall completion time. Results show that 2-exchage heuristic method provides sequences very close to exact solutions in a reasonable time.

Table 3.10 Computational Results (2-exchange improvement procedure).

Problem #	Number of aircraft	FCFS Completion Time (sec.)	Optimal Completion Time (sec.)	2-exchange Sequence	
				Completion Time (sec.)	Computation Time (sec.)*
10_0	10	1883 (100)	1883 (100.0) **	1883 (100.0)	0.010
10_3	10	750 (100)	686 (91.5)	691 (92.1)	0.050
10_4	10	626 (100)	492 (78.6)	492 (78.6)	0.060
10_5	10	644 (100)	518 (80.4)	541 (84.0)	0.050
10_6	10	715 (100)	524 (73.3)	545 (76.2)	0.060
15_0	15	2095 (100)	2095 (100.0)	2095 (100.0)	0.030
15_1	15	1006 (100)	n.a.	956 (95.0)	0.080
15_2	15	948 (100)	n.a.	848 (89.5)	0.110
50_2	50	4670 (100)	n.a.	4602 (98.5)	0.030
50_7	50	3086 (100)	n.a.	3086 (100.0)	6.519

*Computation times do not include times for tour building process. The reported times are CPU seconds on a PC having Pentium Pro 166Mhz CPU and a 64Mega-byte RAM.
**The numbers in parentheses denote the percentage of completion time comparing with FCFS sequence.

From computational experience, it turns out that Rule 1 in Method 1 for enhancement of improvement heuristic does not always provide favorable results. The reason for this is likely that restricting the chances of flipping a segment of the current sequence because of this rule might reduce the possibility for generating a better solution. In fact, the precedence imposed on flights having the same operation type and involving the same aircraft type is not a “hard” constraint but a “soft” constraint in the sense that these precedences may be violated without losing feasibility. While it is advantageous to add these relations to combat symmetry in a mathematical optimization model, in the present context, it is preferable to drop this restriction.

In contrast, the reduction of time-windows is evidently effective in saving computational effort. The computational performance before and after applying the time-window reduction to the same

data set used for Table 3.10 are compared in Table 3.11. In some cases, the computational time increases slightly after applying this reduction. This increase is mainly due to the additional effort for performing this reduction, and appears to be negligible. On the other hand, the number of iterations is reduced significantly in some cases.

Table 3.11 Comparison of Computational Efforts.

Problem #	Before reduction of TW	After reduction of TW
10_0	1 (0.010)*	1 (0.020)
10_3	3 (0.050)	3 (0.050)
10_4	6 (0.060)	7 (0.080)
10_5	10 (0.050)	7 (0.060)
10_6	7 (0.060)	4 (0.040)
15_0	1 (0.030)	1 (0.040)
15_1	5 (0.080)	5 (0.110)
15_2	4 (0.110)	4 (0.130)
50_2	19 (6.519)	17 (6.018)
50_7	1 (0.030)	1 (0.040)

*iterations (CPU time, seconds).

For the purpose of comparison, we also tested the 2-swap heuristic using the same data set that was used for the 2-exchange heuristic. Computational results obtained are summarized in Table 3.12. Compared with the 2-exchange heuristic results (see Table 3.10), the 2-swap heuristic requires much less computation time. It, however, shows that in some cases, the resulting solutions are relatively further from optimality.

Table 3.12 Computational Results (2-swap improvement procedure, n=10).

Problem #	Number of aircraft	FCFS Completion Time (sec.)	Optimal Completion Time (sec.)	2-swap Sequence	
				Completion Time (sec.)	Computation Time (sec.)*
10_0	10	1883	1883 (100.0) **	1883 (100.0)	0.020
10_3	10	750	686 (91.5)	691 (92.1)	0.090
10_4	10	626	492 (78.6)	568 (90.7)	0.040
10_5	10	644	518 (80.4)	562 (87.3)	0.050
10_6	10	715	524 (73.3)	604 (84.5)	0.040
15_0	15	2095	2095 (100.0)	2095 (100.0)	0.020
15_1	15	1006	n.a.	995 (98.9)	0.040
15_2	15	948	n.a.	905 (95.5)	0.100
50_2	50	4670	n.a.	4602 (98.5)	0.220
50_7	50	3086	n.a.	3086 (100.0)	0.971

* Computation times do not include times for tour building process. The reported times are CPU seconds on a PC having Pentium Pro 166Mhz CPU and a 64Mega-byte RAM.

**The numbers in parentheses denote the percentage of completion time comparing with FCFS sequence.

Chapter 4. Network Assignment Problem

The main purpose of the Network Assignment Problem (NAP) is to provide air traffic controllers with efficient taxiing plans that can reduce taxiing delays at the airport under consideration. The plans involve taxiing routes from a gate to a runway (or more precisely, to the departure queue) for departing aircraft, or from the runway exit to a gate for arriving aircraft. At large airports having complex taxiway configurations, taxiing routing is an important task for ground controllers. Even for small airports, this might be important if automated conflict resolution is desired. The simplest way to assign taxiing paths is to establish routes based on shortest paths. Shortest paths are usually static in the sense that the paths are independent of any changes in traffic conditions on the taxiway structure. This approach is relatively easy to implement but is likely to produce more delays. Another option is to apply a time-dependent network assignment strategy which considers changes in traffic conditions over time. In this chapter, network assignment and shortest path algorithms for solving NAP are presented along with computational results.

4.1 Network Assignment Strategies

4.1.1 Network Assignment Algorithm

There are several drawbacks in adopting static assignment techniques as a solver for NAP problems. This mainly because there are differences between urban transportation networks and airfields with respect to the network operations. Most urban transportation links are directed in that they are used in one direction for all days. In contrast, almost all links in airfields are bi-directional in that each link's operational direction can change over time. Whereas urban transportation networks are directed graphs, airfield networks are undirected graphs in their network representation. In order to consider time-dependent characteristics of airfield links, dynamic (or time-dependent) network assignment strategies which have a look-ahead function for each link's operational direction should be introduced.

In a time-dependent network assignment strategy, all aircraft interact as long as they overlap within the time horizon. By rerouting all vehicles iteratively, a dynamic user equilibrium (*UE*) (or system optimal (*SO*)) state can be achieved. The general condition describing a dynamic *UE* can be expressed as follows [Ghali, 1995]:

If, at each instant in time, for each origin-destination pair, the unit costs of flow on utilized paths are identical and equal to the minimum instantaneous unit path cost, the corresponding flow pattern is said to be user optimized.

In this research, the tentatively named *quasi-dynamic network assignment strategy* is proposed. In a quasi-dynamic network assignment strategy, it is assumed that the current aircraft route is influenced only by the previous aircraft assigned to the network. This simplified assumption rules out the necessity of iterative rerouting procedures, thereby reducing the number of computations.

Besides the computational aspect, there is another theoretical rationale behind using the quasi-dynamic network assignment strategy. In the *UE* (or *SO*) framework, it is assumed that vehicle drivers select their best paths in such a way that their own travel times (or marginal travel time) are minimized. It is also assumed that all drivers have equal priorities in the sense that the rule of first-come-first-served is maintained during travel. At an airport, however, ground controllers often provide higher priorities to certain taxiing aircraft over others to facilitate traffic flows. This is a non-systematic situation where it might be impossible to achieve a pure dynamic *UE* (or *SO*) state.

Figure 4.1 depicts the flowchart for a quasi-dynamic network assignment method. The method is based on the incremental assignment technique which decides the aircraft path in a one by one fashion. After an aircraft is routed, the links on the taxiing path are loaded by that aircraft over time slices (see Figure 4.2). As mentioned earlier, once any aircraft is assigned to taxi on a link in one direction, the opposite direction of the link should be blocked during the time slice occupied by the aircraft to prevent any conflict on the taxiway. This can be done by increasing the travel time of the conflicting link to an artificially large number to avoid being selected as a time-dependent shortest path for any other flight. In practice, all the information about the loaded aircraft is maintained in an fixed size array, called a *time-dependent aircraft flow table*. Figure 4.2 shows the resulting time-dependent link flows and the time-dependent link travel times for the case that a flow v_1 traverses from O to D along the path O→1→2→D.

The time-dependent travel times for all links can be computed by applying any known link performance function link traffic volumes. Once the time-dependent link travel times are updated, the time-dependent shortest paths between all nodes are recomputed for next flight to be assigned using a time-dependent shortest path algorithm.

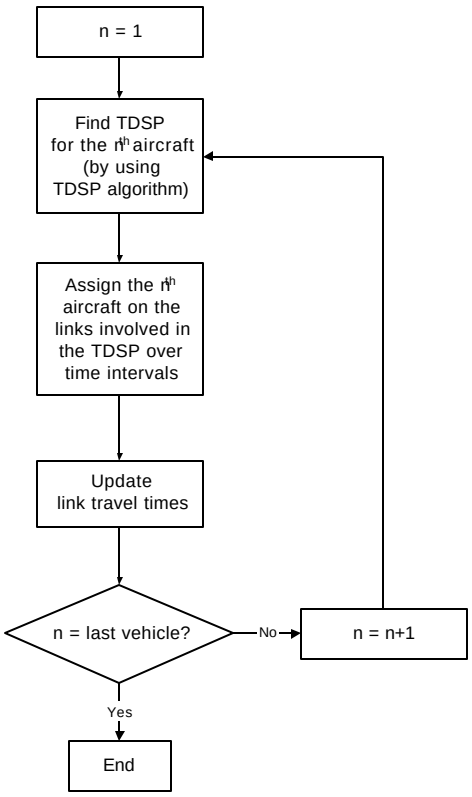
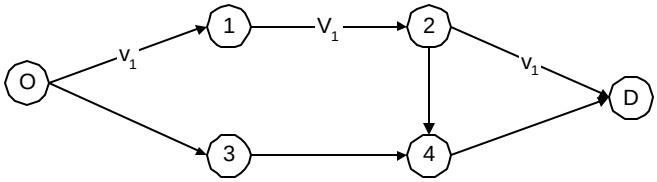


Figure 4.1 Quasi-dynamic Assignment Algorithm (1).



Time-dependent vehicle flow

Time slices \ Links	L_{O1}	L_{1O}	L_{O3}	L_{3O}	L_{12}	L_{21}	L_{34}	L_{43}	L_{24}	L_{42}	L_{2D}	L_{D2}	L_{4D}	L_{D4}
1	v_1	-	-	-	-	-	-	-	-	-	-	-	-	-
2	-	-	-	-	v_1	-	-	-	-	-	-	-	-	-
3	-	-	-	-	-	-	-	-	-	-	v_1	-	-	-

Time-dependent link travel times

Time slices \ Links	L_{O1}	L_{1O}	L_{O3}	L_{3O}	L_{12}	L_{21}	L_{34}	L_{43}	L_{24}	L_{42}	L_{2D}	L_{D2}	L_{4D}	L_{D4}
1	t_{O1}^{1}	∞	-	-	-	-	-	-	-	-	-	-	-	-
2	-	-	-	-	t_{12}^{2}	∞	-	-	-	-	-	-	-	-
3	-	-	-	-	-	-	-	-	-	-	t_{2D}^{3}	∞	-	-

Figure 4.2 Quasi-dynamic Assignment Algorithm (2).

4.1.2 Shortest Path Algorithms

The time-dependent shortest path algorithm suggested by Ziliakopoulos and Mahmassani is designed to provide time-dependent shortest paths for all O-D pairs every time slices. In terms of problem size, the NAP on taxiway system is much smaller than the NAP on the urban transportation network. For example, the number of nodes and links on taxiway system at a large airport is much less than that on a typical urban network. Also the number of aircraft considered in the airport is smaller than the number of cars in the urban network. Motivated by this point, we employ a TDSP algorithm based on the Dijkstra's algorithm having the following characteristics:

1)

The TDSP provides a time-dependent shortest paths from a single root node to all other nodes starting at time t.

The TDSP algorithm uses the sorted-queue as a data structure for candidate nodes, which makes the algorithm as LS algorithm rather than LC algorithm. (Even though the size of candidate nodes set (i.e., SE list) is small, size are big, fast sorting algorithm such as "quick sort" can be applied for the facilitating the sorting procedure.)

Figure 4.3 shows the suggested TDSP algorithm for time-dependent paths from a node, r, to all other nodes starting at time t.

```
Compute CurrentTimeSlice.

Call Initialize

while(SE list is not empty)
        u = deQueueMin(r);
        for(v = all forward star of u){
                d_u = shortTime(r, u);        // travel time from r to u.
                d_v = shortTime(r, v);         // travel time from r to v.
                Compute timeSliceFor_l_uv;//compute the time slice for travel time from u to v.
                l_uv = Find travelTime(u, v, timeSliceFor_l_uv)
               // find the travel time from u to v for timeSliceFor_l_uv
                if(d_v > d_u + l_uv){
                        shortTime(r, v) = d_u + l_uv; // update the short time from r to v
                        enQueue(v);
                }
        } //end for
}// end while

Procedure Initialize:
        for (i = all nodes){
            predecessor(i) = 0;
```

```
    for (j = all node){
        if(i ≠ j) Label(i, j) = inf.;
        else Label(i, j) = 0;
    }
  }
predecessor(r) = r;
```

Procedure deQueueMin(): // find the closet node from the SE list.

Procedure enQueueMin(x): // insert node x into the SE list.

Figure 4.3 Pseudo Code for Suggested Time-dependent SP Algorithm.

4.2 Computational Results

To compare static and time-dependent assignment strategies, let us consider a hypothetical flight schedule for DCA airport. (For the airport configuration, see Figure 5.3. The details of this airport are described in Chapter 5.) As shown in Table 4.1, the schedule has only two flights but it is designed to have a conflict during taxiing to their own destinations.

Table 4.1 Hypothetical Flight Schedule.

Flight #	Aircraft Type	Operation Type	Schedule Time			Gate	Runway
			Hour	Minute	Second		
DEP_1	B727-100	Departure	0	0	27	1	36
DEP_2	B727-100	Departure	0	0	44	11	16

Figure 4.4 shows the paths resulting from two different assignment strategies. Since no flight is on the taxiway when the flight "DEP_1" is assigned, it can taxi to its destination at unimpeded (or free-flow) speed. Therefore there is no difference in static and time-dependent shortest paths for flight "DEP_1".

In the static assignment strategy, "DEP_2" taxis and approaches "DEP_1" on the link (1020, 1024). On the other hand, for the time-dependent assignment case, "DEP_2" is assigned to make a detour so as to avoid the conflict with "DEP_1". This is obvious because, when "DEP_2" is assigned on the network, link (1024, 1020) has been blocked by "DEP_1", and the time-dependent SP algorithm detects this blockage. The difference of two paths for "DEP_2" are clearly illustrated in Figure 4.5.

Paths from Static Network Assignment

Flight "DEP_1": 1 (27') - 1009 (44') - 1011 (49') - 1014 (54') - 1016 (65') - 1017 (77') - 1019 (83') - **1020 (90') - 1024 (117')** - 1027 (126') - 1031 (143') - 2018 (171') - 2021(202').

Flight "DEP_2": 11 (44') - 1029 (52') - 1030 (61') - 1026 (70') - **1024 (78') - 1020 (105')** - 1021 (111') - 1018 (122') - 1015 (140') - 1012 (149') - 1010 (157') - 1008 (182') - 1005 (196') - 1002 (235') - 1001 (250') - 2001(261').

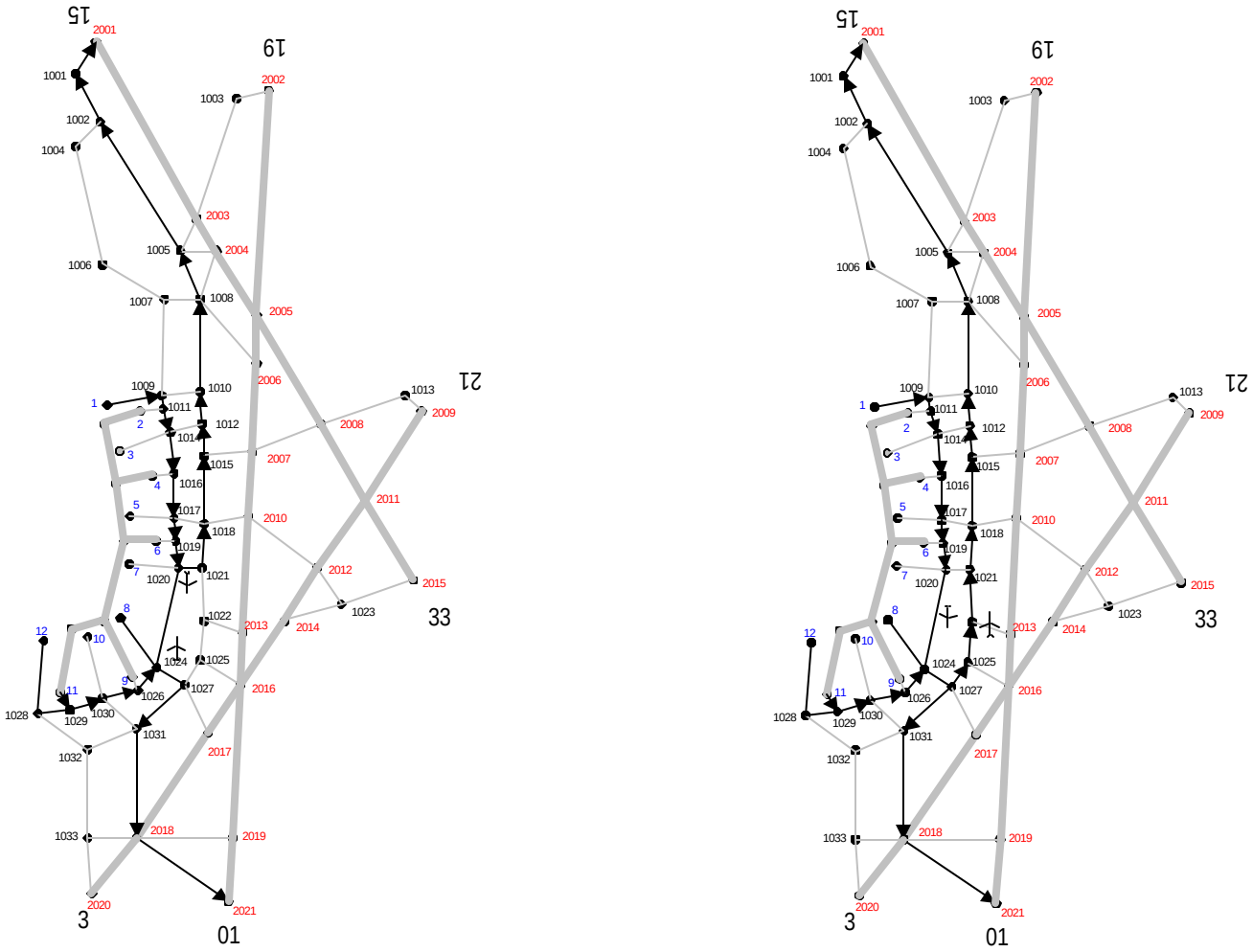
Paths from Time-dependent Network Assignment

Flight "DEP_1": Same as Static Path.

Flight "DEP_2": 11 (4') - 1029 (52') - 1030 (61') - 1026 (70') **- 1024 (78') - 1027 (87') - 1025 (94') - 1022 (105')** - 1021 (119') - 1018 (130') - 1015 (148') - 1012 (157') - 1010 (165') - 1008 (190') - 1005 (204') - 1002 (244') - 1001 (258') - 2001(269').

Legend: Node (Seconds).

Figure 4.4 Static vs. Time-dependent Path (1).



(a) Statically assigned path.

(b) Time-dependently assigned path.

Figure 4.5 Static vs. Time-dependent Path (2).

Chapter 5. Virginia Tech Airport Simulation Model

In this research, the simulation model plays an important role as a supplement to the analytical models developed for ASP and NAP. This is because the simulation model makes it possible to portray the dynamic behavior of entities that are difficult to examine by analytical models. In this Chapter, methodologies pertinent for developing our simulation model are reviewed along with previous simulation models. We also present detailed procedures to develop a simulation model named Virginia Tech Airport Simulation Model (VTASM).

5.1 Framework of the Proposed Simulation Model

The primary purpose of a simulation model in the NAP is to evaluate the current state of aircraft in the taxiway and runway systems. Several measures of network performance can also be estimated using the simulation model. These include the estimation of delays and a general assessment of the network congestion.

Since the number of entities (aircraft and air traffic controllers) considered in airport simulation models is relatively small, a discrete time, microscopic and deterministic simulation model is considered appropriate for this research problem. Even though the current prototype model is developed as a deterministic model, the model design permits the inclusion of random variables to reflect the stochastic behavior observed at airport networks.

The development of the simulation model is based on standard object-oriented methods commonly used in software engineering. The most prominent qualities of an object-oriented model are: 1) an easier understanding of the system, and 2) modifications to the model tend to be local as they often result from an individual item, which is represented by a single object [Jacobson, 1992].

Object-oriented methods have been improved in the past decade with the development of many new object-oriented programming languages. The number of object-oriented methods increased from less than 10 to more than 50 during the period between 1989 and 1994. Among the methods, the most notable include Booch and Jacobson's OOSE (object-oriented software engineering), Rumbaugh's OMT (object model technique), Shaer-Mellor's method, and the Coad-Yourdan's method. Recently the "Unified Modelling Language (UML)" has been developed by Booch, Jacobson and Rumbaugh to unify various object-oriented methods [Booch et al., 1998].

The simulation model devised in this research project has been developed by following an extension of the OMT originally applied to the design of SIMMOD 3 [ATAC, 1993]. In general, the object-oriented modeling process is decomposed into three phases: 1) Object-oriented Analysis (OOA), 2) Object-oriented Design (OOD) and 3) Object-oriented Programming (OOP). Figure 5.1 describes the general procedures of the OMT methodology [Weijers, et. al, 1995 and Derr, 1995].

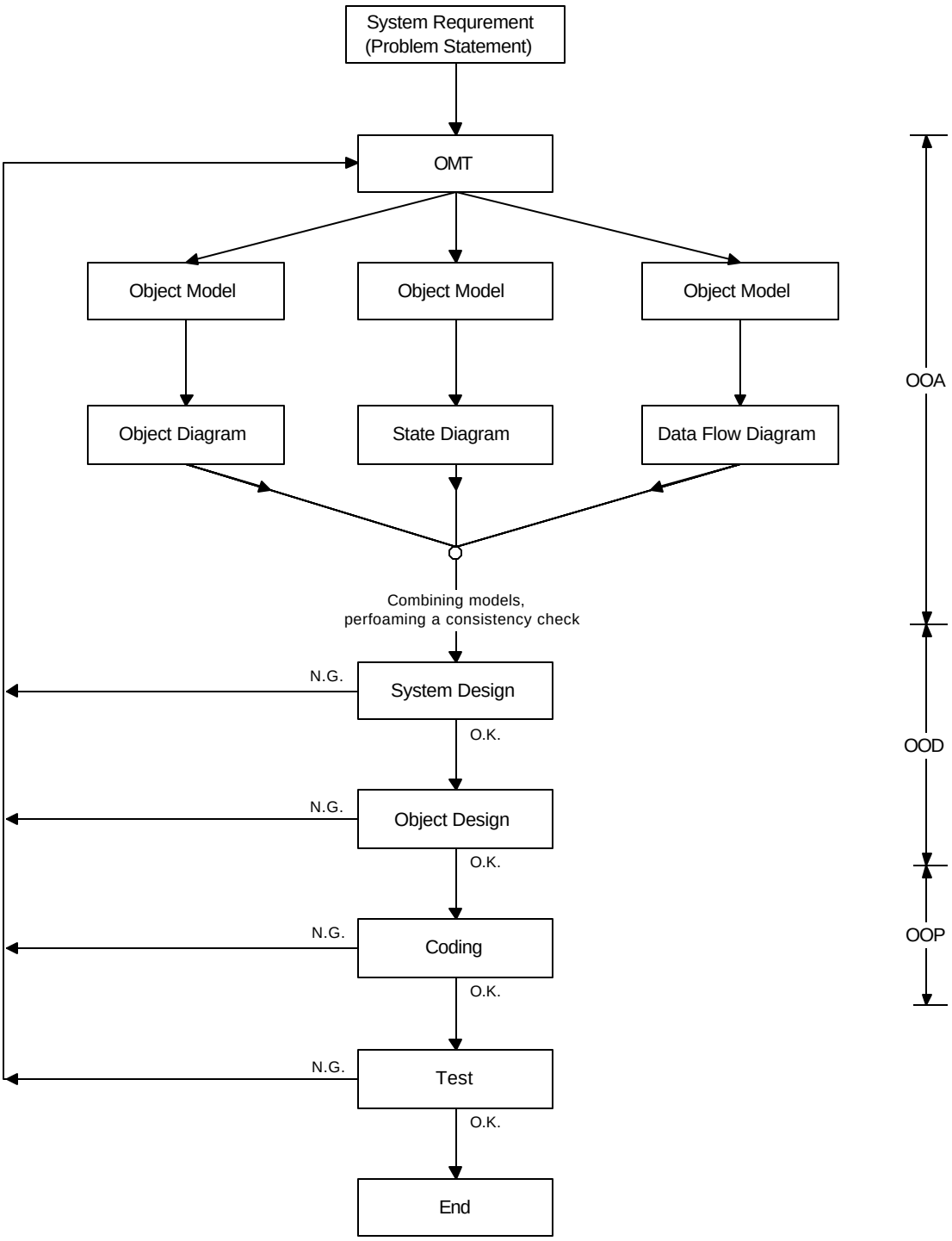


Figure 5.1 Processes of OMT [reproduced from Weijers et. al, (1995) and Derr (1995)].

5.2 Object-oriented Analysis (OOA) of the Simulation Model

OOA can be defined as a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain [Booch, 1994]. OOA begins with the verbal description of a problem statement. Then three types of models are constructed: an *object model*, a *dynamic model*, and a *functional model*.

The *object model* describes the static, structural, and data-aspects of objects in the system, along with their inter-relationships. The object model is represented by object diagrams. An object diagram is a graph whose nodes are object classes and whose arcs reflect the relationships among classes. The *dynamic model* describes the interactions among objects in the system in terms of state diagrams. These represent the temporal, behavioral, and control-aspects of the system changing over time. In a state diagram, nodes represent states, and arcs represent transitions between states.

The data value transformation within a system is described in the *functional model*. Functional models include data-flow diagrams where nodes represent processes and arcs represent data flows [Weijers, et. al, 1995]. In the end, the three models come together during implementation, which involve data (object model), sequencing (dynamic model), and operational (functional model) manipulations [Bakker et. al, 1995].

5.2.1 Object Model

The main focus of the object model is to construct an object diagram that describes the data-oriented static structures of the problem domain. The following steps are used in the object model analysis according to the OMT method [Rumbaugh et al., 1991, and Derr, 1995]:

Step 1. Develop a problem statement.

Step 2: Identify the object classes which represent all the physical and conceptual objects from the problem statement.

Step 3. Prepare a data dictionary giving a short description of various entities in the model.

Step 4. Identify associations (including aggregations) between object classes.

Step 5. Identify attributes of the object classes.

Step 6. Use inheritance to share common structures.

Step 7. Traverse access paths using scenarios to identify deficiencies.

The candidates of objects or classes may be found by identifying noun or noun phrases from the problem description. Association is a relationship and represents dependencies between classes, being usually denoted by verb or verb phrases in the problem statement. Attributes are the data values that are held by the objects, and are usually imbedded in a noun followed by possessive phrases [Derr, 1995].

The arrow in the object diagram shows an aggregation/association relationship. An object at the tail of arrow has an object at the head of arrow. The airport network, for instance, has *n* nodes and *m* links and each link consists of two nodes defined as the from-node and the to-node. A gate is represented by a node, and each node can hold one aircraft (or *n* aircraft in the case of an aggregate gate). A link can have at most *n* taxiing flights at the same time.

Each flight has a taxiing path which consists of a series of links from the gate to the runway for departing flights (or vice versa for arriving ones). Aircraft can have *n* flight schedules. Controlling flights at an airport is carried cooperatively by two types of air traffic controllers: 1) a ground controller, and 2) a local controller. Air traffic controllers evaluate various flight requests to pushback, taxi, takeoff, or land, and try to maintain a smooth flow on the taxiway and runway infrastructure while minimizing delays. Even though the final decisions for the ground and local controllers are different, they share common attributes and behavioral characteristics. For example, both controllers have the same information on flights, airport configuration, and use the same communication procedures etc. For this reason, the ground and the local controllers are inherited from the common class called *controller*.

Clock object is of importance in the simulation model as it is used in scheduling the next event for such entities as controllers and flights. The object diagram is illustrated in Figure 5.2. The various class definitions are described in Appendix C.

5.2.2 Dynamic Model

The time-dependent aspect of each entity is represented in the so called state diagram (or state transition diagram) which consists of states and events. A change of state caused by an event is called a transition. Whereas an object model describes the possible patterns of objects, attributes, and links in a system, a dynamic model represented by the state diagram highlights the time-dependent behavior of a system [Weijers, et. al, 1995]. According to Rambaugh (1991), the general steps for constructing state diagrams are as follows:

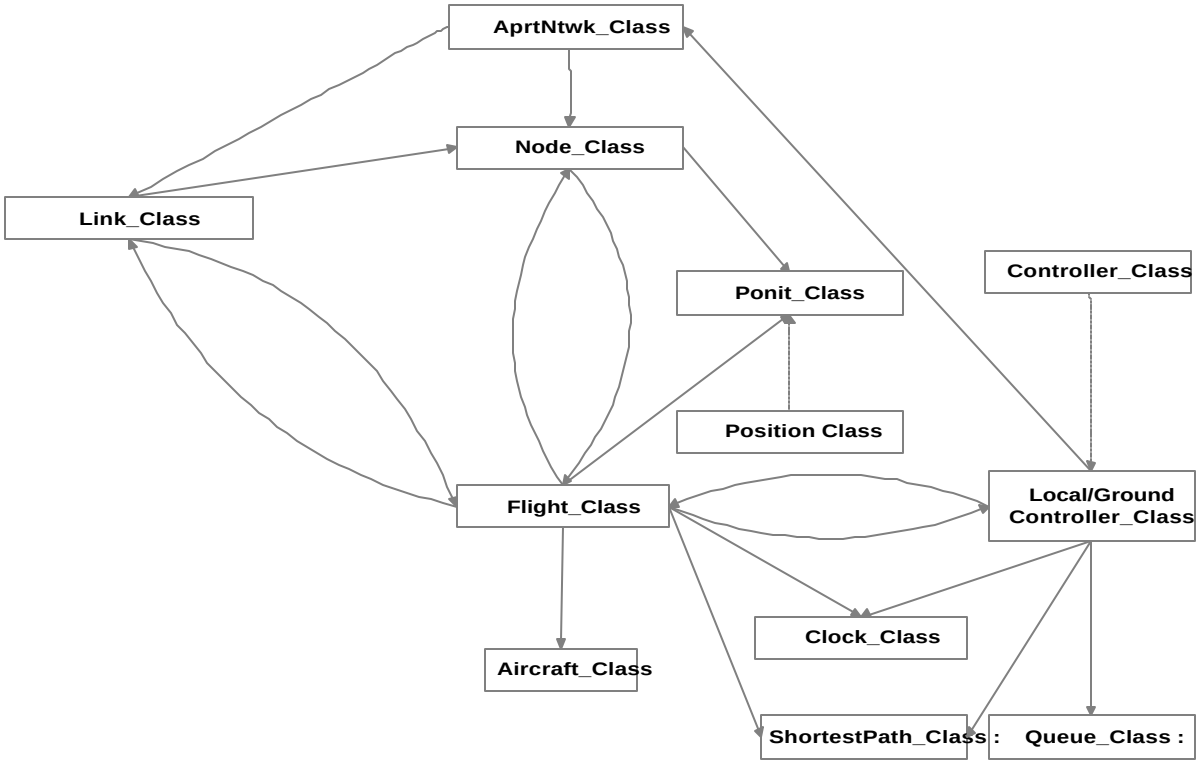


Figure 5.2 Object Diagram.

Step 1: Prepare scenarios for typical interaction sequences.

Step 2: Identify events between objects and prepare an event diagram for each scenario.

Step 3: Build a state diagram.

Step 4: Match events between objects to verify consistency and completeness of events shared among the state diagrams.

5.2.2.1 Arriving Flights

Arriving aircraft are introduced to the simulation at discrete times prescribed by the original flight schedule. If a flight scheduled landing time violates the minimum separation headway criteria, the flight is delayed at the entry node (the runway interface). This procedure models implictly the fina; airspace corridor leading to the active runway.

A. State Diagram for Ground Movement

Once an arriving flight obtains landing clearance from the local controller, it begins its landing roll procedure which is composed of four phases: flaring out, free rolling, braking, and coasting [Trani et. al, 1993]. After the completion of these four phases, the pilot decides which exit to use by considering both the current speed and the location of neighboring exits. (A more detailed landing procedure will be discussed later). Once a flight starts to exit from the runway, it will contact the ground controller for taxiing clearance instructions.

The aircraft movement on the taxiway system is modeled regulating the vehicle acceleration to accomplish a smooth and safe movement to the gate. This procedure is executed in an autonomous fashion as the aircraft moves along the given path, keeping a safe spacing with other taxiing flights. For example, the leading aircraft movements are modeled by a second-order feedback control system to be discussed later in this chapter. The state transition diagram for an arriving flight is depicted in Figure 5.3.

B. State Diagram for Communication

A unique aspect of VTASM is that air traffic controllers are modeled as explicit resources. If the corresponding controller is busy communicating with another flight when the current flight tries to contact the controller, the current flight waits for a certain period of time until the controller state turns to standby.

Here, the waiting process varies depending on the communication system. In this research model, two types of communication systems are considered: 1) a *voice channel system* which has been widely used in aviation for seventy years, and 2) a *data link system* which is an advanced system using electronic data transmission so the time required for either sending requests or receiving command is considered to be negligible.

Regardless of communication system type, the flight's communication state is initially set to "readyToCommunication". Once the flight finds the need to communicate with a controller, then it tries to send a request, which will be accepted by the controller unless the controller is busy.

If the controller is busy, it happens under the voice channel system that the flight's state is set to "waitNextCommunication" and the flight is required to wait for the next contact for a certain amount of time. The operation of the data link system follows smilar principles. The flight's state changes to "waitContactFromController" and the flight waits controller's contact.

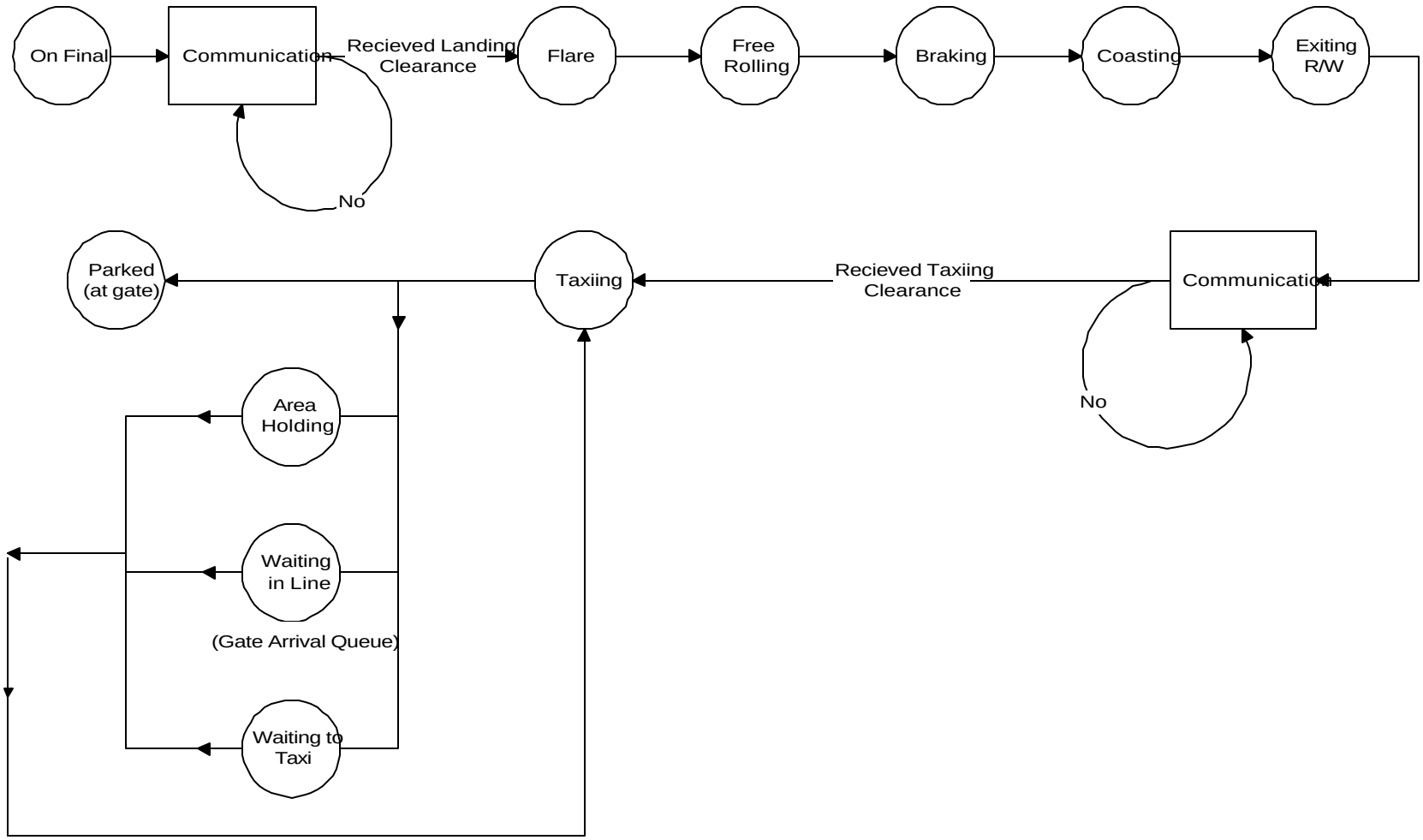


Figure 5.3 State Diagram for an Arriving Flight.

If the controller is in a state of "standby" when the flight attempts a contact, the controller's state is set to "busy". Once a flight succeeds in contacting the controller, it communicates exclusively with the controller until the entire communication phase is completed. After the flight confirms the command, the flight's and controller's state are set to "readyToComm" and "standby", respectively.

Depending on the traffic congestion levels on the taxiway system, it is possible for the flight to be delayed even after communicating with a controller. In this case, the flight communication state is set to "wait for controller's contact". The state transition diagrams for both communication systems are depicted graphically in Figures 5.9 and 5.10.

5.2.2.2 Departing Flight

A. State Diagram for Movement

A departing flight stays in the "parked" state until the scheduled departure time arrives. During the last one or more hours of this state, a departing flight files a flight-plan with the FAA flight service station (FSS) and loads passengers or freight. Having finished all required routines, the flight contacts the ground controller for the pushback clearance. The pushback clearance is frequently given to the flight along with a taxiing clearance by a local controller. However, in some large-sized and busy airports, the pushback process is controlled by the gate manager. The pushback is the initial process of departure in which the flight trespasses the apron area from the gate to the taxiway.

Once the aircraft arrives at the taxiway from the gate with a taxiing clearance, a flight starts taxiing to the designated runway. (The flight needs to contact the ground controller if it has not acquired a taxiing clearance.) The taxiing procedure is almost the same, but in reverse, to the taxiing process for an arrival. The flight autonomously traverses the taxiing path given by a ground controller. Usually, the (ground) controller does not interrupt a flight taxiing schedule, unless there is some new expected congestion or conflict. If a controller anticipates congestion, he or she can give the flight a command to slow, stop, or change the taxiing path. The goal of this interruption is to reduce the congestion on the taxiway network and, at the same time, maximize the utilization of the runway.

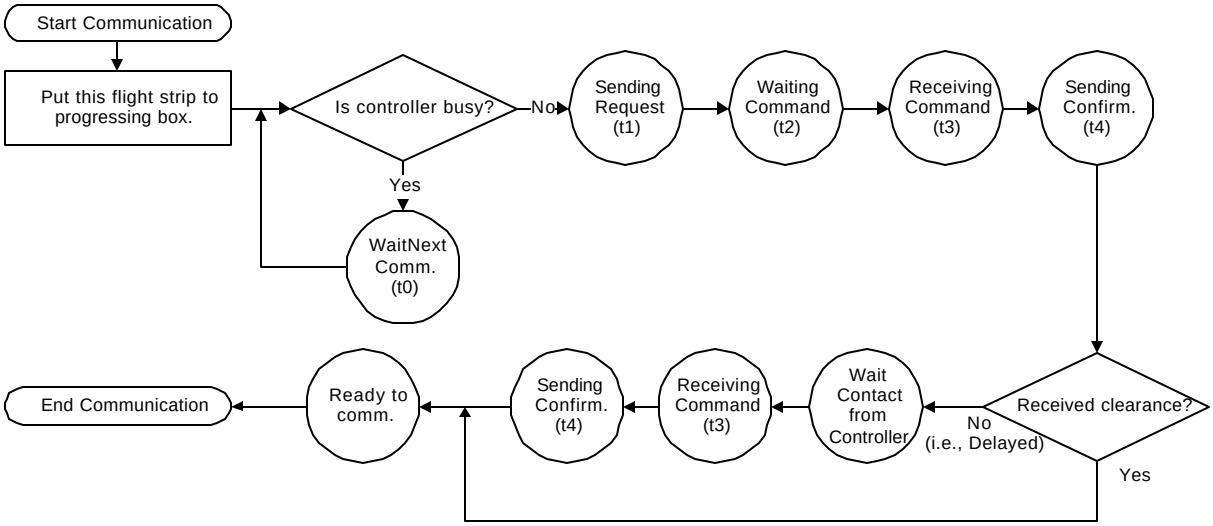


Figure 5.4 State Diagram for Communication (Voice Channel).

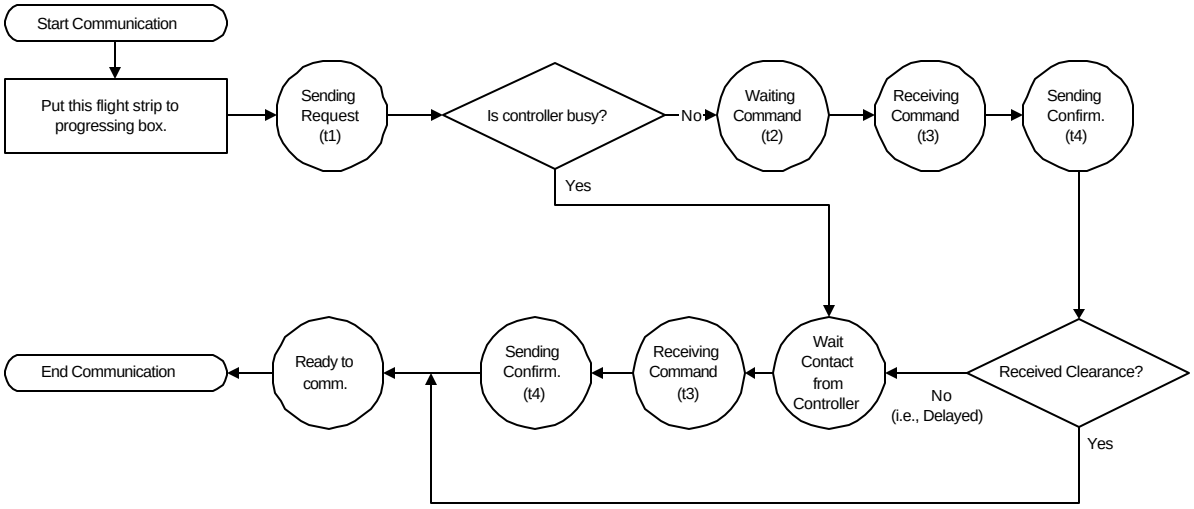


Figure 5.5 State Diagram for Communication (Data Link).

When the flight approaches the active runway, it is required to reduce its speed and contact the local controller to obtain takeoff clearance. Note that the previous taxiing clearance is not a clearance to enter onto the runway. If there is a queue of flights waiting for takeoff, the local controller gives the takeoff clearance based on a first-come-first-served policy, ensuring a particular airborne separation by spreading out the aircraft departure intervals [Luffsey, 1990].
The takeoff process is composed of two processes: 1) rolling process in which the flight gains the speed required for lift-off, and 2) the lift-off process in which the aircraft starts to climb and clears the far end runway threshold. In the simulation model, the departing flight is traced until it reaches the runway threshold. The state diagram for a departing flight is illustrated in Figure 5.6.

B. State Diagram for Communication

Basically, the departing flight state diagram for communication is same as that of an arriving flight.

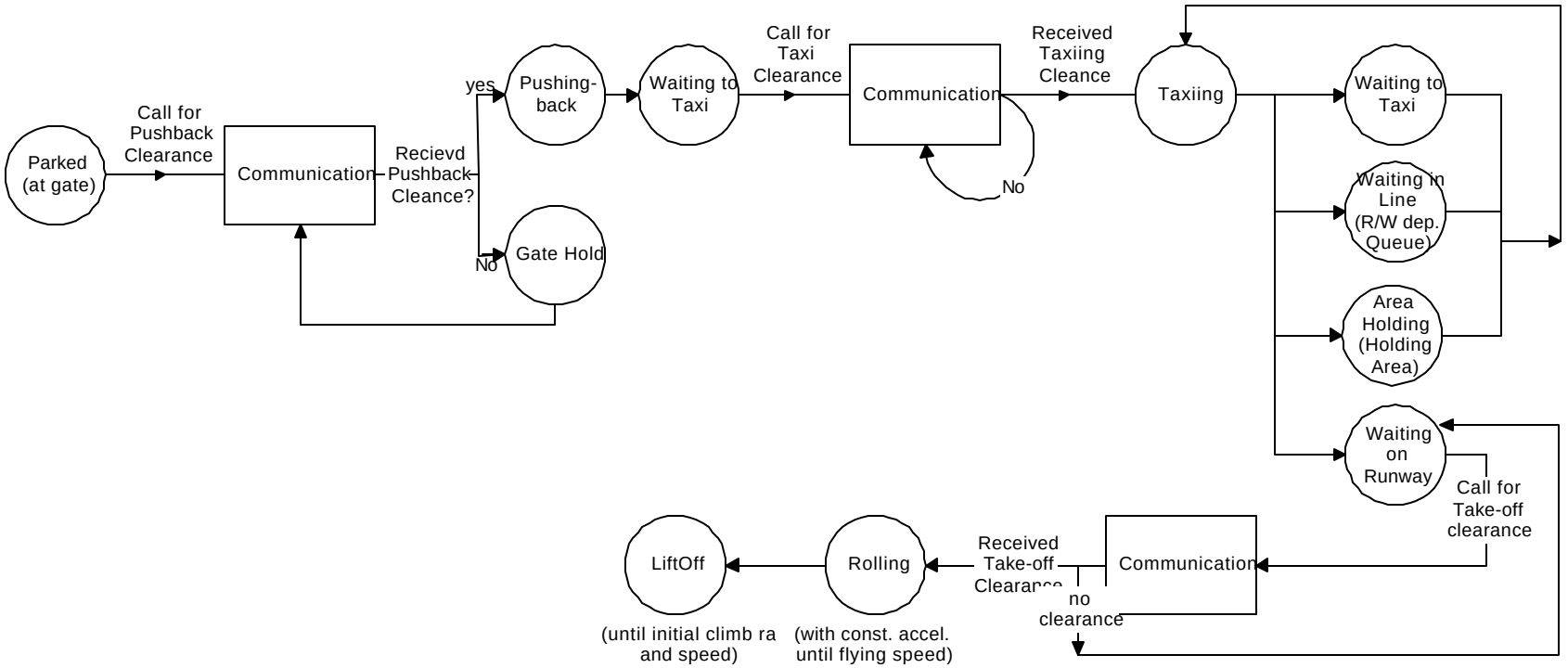


Figure 5.6 State Diagram for a Departing Flight.

5.2.2.3 Controller

Depending on who initiates the communication, air traffic controlling processes can be divided into two classes: 1) *passive control* in which the controller renders control messages to various flight requests, and 2) *active control* in which the controller makes decisions mainly for controlling the overall traffic flow. In the process of air traffic controlling, *flight progress strips* (or flight strips for short) are used to store critical flight information such as flight number, aircraft type, origin, destination or arriving route, etc. This information is printed in a rectangular piece of paper, and stored in a plastic holder while the flight is under supervision of an air traffic controller. In the simulation model, three type of flight strips are modeled: pending, processing and completed flight strips (See Figure 5.7).

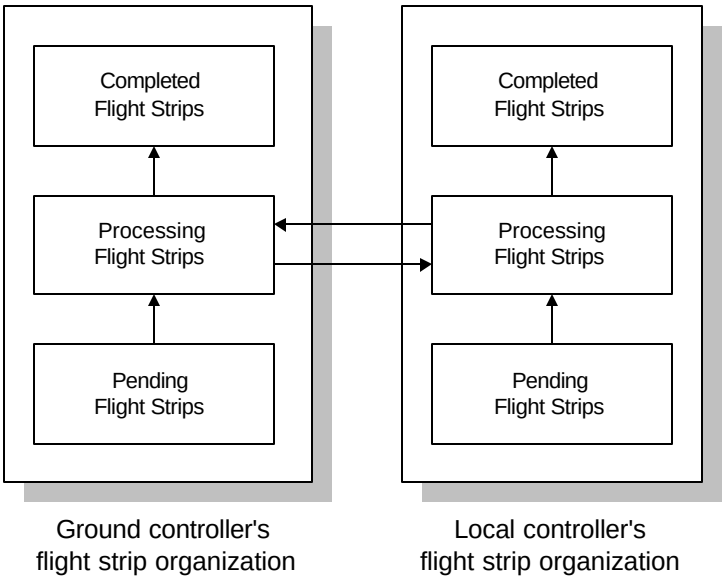


Figure 5.7 Flight Progress Strip.

Initially, the controller's state starts with "standby" and all flight strips are in the pending box. Receiving a request from a flight in the passive control mode, the controller places the corresponding flight strip in the processing state and begins to judge the situation. The controller's judgement depends on the flight's current state. For example, if a flight is at the final stage in the arrival process and it requests landing clearance, the controller needs to check the flight's relative position around other flights, and then make a decision on whether or not to allow

this flight to proceed (checking for minimum separation rules between flights). The controller sends control messages to the flight and waits for confirmation. Once a confirmation is received from the flight, the controller returns to the "standby" state and waits for another request. The state transition diagram for the controller is depicted in Figure 5.8.

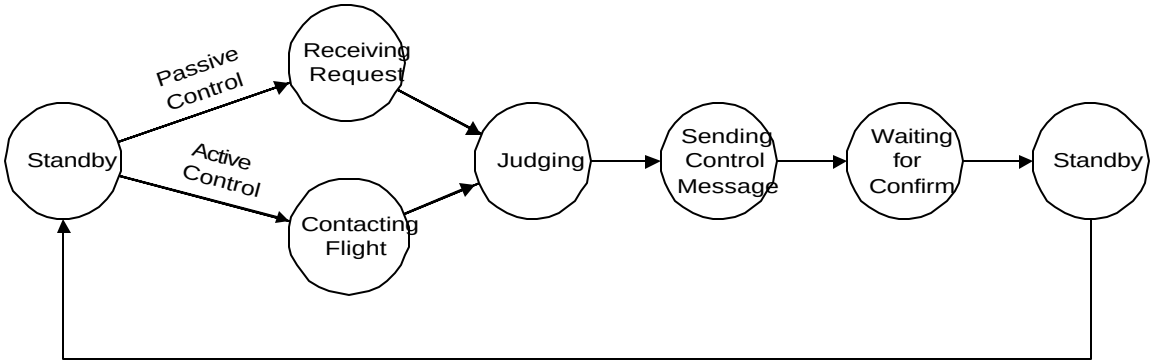


Figure 5.8 State Diagram for a Controller.

When the flight finishes its operation completely, the controller moves the flight strip from the processing to the completed state. If a flight is still performing some activity but passes the limits of a controller's custody (control boundary), then the current controller hands the fight strip over another controller's processing state bin. For instance, if a flight exits the runway taxiing after completion of the runway landing roll and starts taxiing to gate, then the flight strip moves from the local controller's processing bin to the ground controller's processing bin.

If any traffic congestion or conflict is expected, controllers can intervene during aircraft taxiing, issuing control messages to slow-down, speed-up, or even to stop aircraft at the current position. This active control decision making process is largely based on the controller's experience, subject to ATC rules enforced by the FAA. It is difficult to devise a single comprehensive traffic management rule that is applicable to all ground control cases. For the simulation purpose, it might be more practical to develop a rule-based decision making process for the active control process.

A controller also initiates an active control process when the controller is in "standby" and there is some flight in the state of "wait for controller's contact". In fact a controller realizes the existence of flights awaiting controller's contact by checking the strips in the processing bin. The communication process initiated by controller is shown in Figure 5.9.

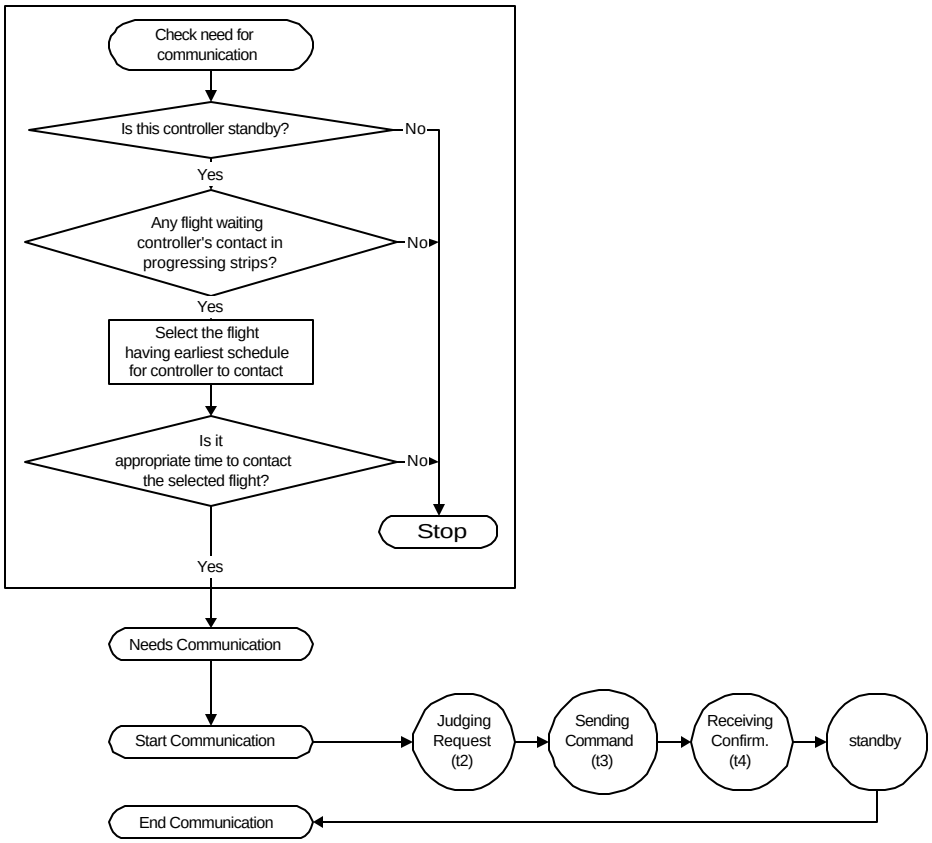


Figure 5.9 Communication Process Initiated by the Controllers.

5.2.3 Functional Model

The functional model consists of Data Flow Diagrams (DFDs) and defines processes within a system, describing how output values of the program are derived from the input values. Later, DFD is used to determine which objects send messages to other objects. DFD is composed of three components: process, data flows actors, and data stores which are represented in the OMT methodologies by ellipsess, rectangles and pairs of parallel lines, respectively. The OMT specifies the following steps for constructing a functional model [Rumbaugh et al., 1991].

Step 1: Identify input and output values.
Step 2: Build data flow diagrams for each input to output transformation.
Step 3: Develop descriptions for each process in the DFDs.
Step 4: Identify constraints between objects.

Step5: Specify any optimization criteria, e.g., values to be maximized or minimized.

Figure 5.10 illustrates the input and output values of the simulation model at the highest level of abtraction. Here, the user interface which is devised for users to control the program progress is not included. This is consistent with the research nature of this model. The main goal of this research poject is to develop a comuptational engine that drives the simulation model. There are two kinds of inputs for the simulation model: 1) scenario data including the flight's schedule, aircraft types, gate information, etc., and 2) static data including the airport configuration, an aircraft model data with aircraft dimensions, performance, etc. All these parameters remain unchanged throughout the simulation.

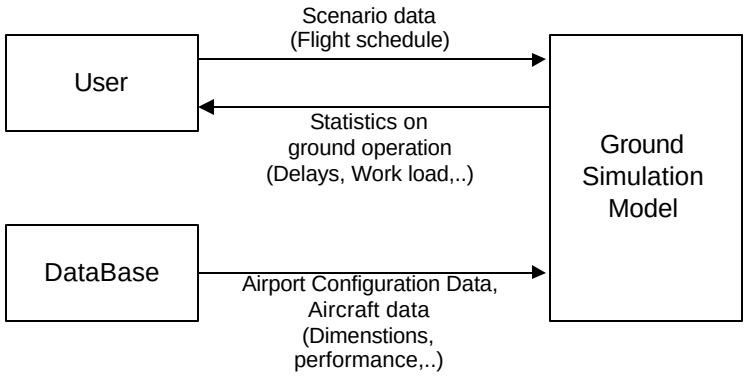


Figure 5.10 Input and Output Values for the Simulation Model.

The top-level DFD for the simulation model is shown in Figure 5.11. The flight schedule information provided by the user serves as the input data for generating the flight objects along with the aircraft model data. The link and node data pertinent to the airport configuration is used to generate the airport network graph, which in turn is used to define the flight taxi path information. The controller objects are created internally, which means that no explicit input data is involved in generating the controller object. Instead, the objects belonging to the airport graph class and the flight class are imbeded in the controller objects.

The DFDs are typically shown as layered sets of diagrams because they are generally too large to be shown on a single piece of paper. The decomposition of DFD continues until the processes cannot be partitioned any further. Because of their complexity, lower-layered DFDs are omitted here.

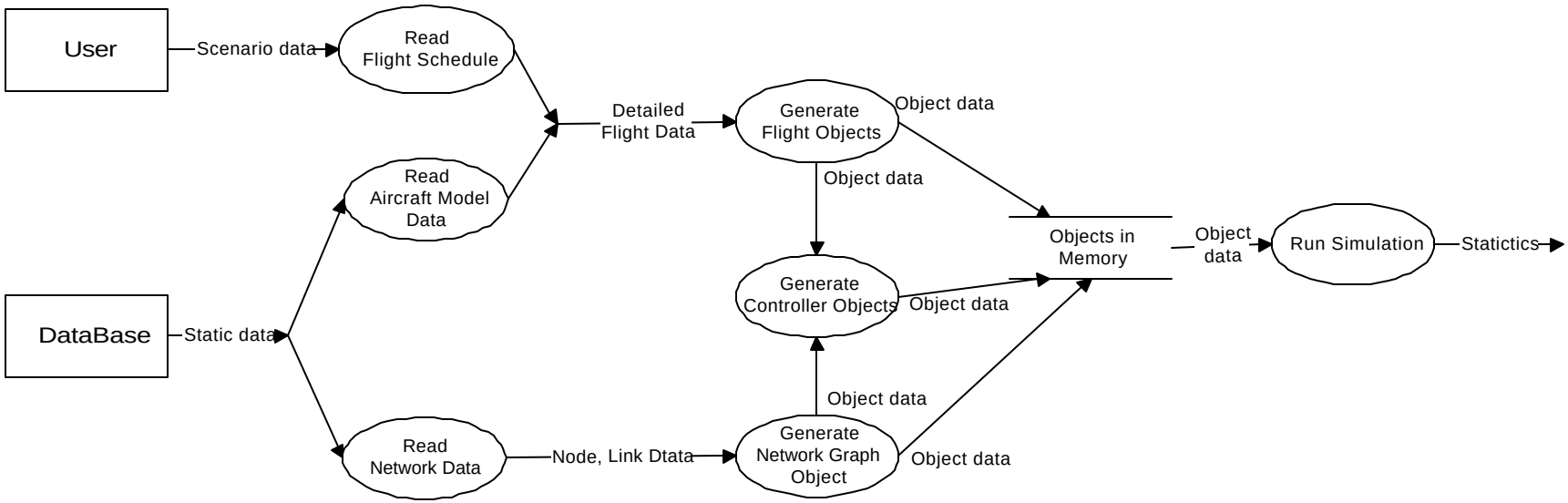


Figure 5.11 Top-Level of Data Flow Diagram (DFD) for VTASM.

5.3 Object-oriented Design (OOD) of Simulation Model

5.3.1 System Design

In system design, the strategic decisions are made at a high level about how the problem will be solved. The system design includes finding answers to the following questions: 1) what components libraries, database, networking mechanisms, windowing systems will be used, and how should they be used; 2) how will tasks communicate; 3) how will tasks be allocated to processors; 4) what is the target environment for the application; 5) will the application need to run on multiple platforms, and 6) what programming language will be used to implement the design [see Derr, 1995].

The main objective of this research is to develop the backbone of a simulation model that considers decisions in response to questions 1, 2, and 3 which are directly related to the software implementation. For code portability, the American National Standards Institute (ANSI) C++ is used to address truly muti-platform compatibility. From the same reason, wxWindows is preferable windowing libraries to any commercial ones such as MFC (Microsoft Foundation Classes).

5.3.2 Object Design

Object design is concerned with a full specification of the classes, associations, attributes, and methods necessary for implementing a solution to the problem. Algorithms and data structures are also fully defined along with any internal objects needed for the implementation. The following steps are typical in an object design phase [Rumbaugh et al., 1991 and Derr, 1995].

Step 1: Identify methods for each class which define all the required types of functions.

Step 2: Design algorithms to implement methods by choosing efficient ways of coding algorithms, selecting appropriate data structures, and defining new internal classes and methods.

Step 3: Optimize access paths to data by adding association classes to minimize access time, rearranging the processing order of algorithms for efficiency, and saving calculated data to avoid re-calculations.

Step 4: Design a method for dynamic control by refining the methods for implementing the flow of events in the dynamic model.

Step 5: Adjust class structures to increase inheritance.
Step 6: Design object associations.
Step 7: Represent object data.
Step 8: Hide data and package classes into modules.

For Step 2, standard flow charts are adopted to present the relevant algorithms. Traditionally, three types of diagrams are used in some applications of OMT. These are the message hierarchy, events trace, and object interaction diagrams (Derr, 1995) which are not considered in this research. During the implementation, the design is translated into code using an object-oriented programming language such as C++.

5.3.2.1 Algorithms

5.3.2.1.1 An Algorithm for Aircraft-following Behavior

As previously reviewed, the generalized vehicle-following model presented in Equation (2.2) can be used to predict the acceleration (or deceleration) at time t for a following vehicle that considers the kinematic state of a leading vehicle. The application of vehicle-following equations yields various types of macroscopic traffic flow models (see Table 2.9).
A well-known car-following model used in transportation studies is described in Equation (5.1), where the coefficient m and l are 0 and 2, respectively. The macroscopic version of this model is known as the Greenshields' model, and is one of the most frequently used models in traffic engineering studies. Greenshields' model has significant advantages, namely, that it is simple to use and has shown good correlation between the model and field data.

$$a_{n+1}^{t+\Delta t} = a\,\frac{(v_n^t - v_{n+1}^t)}{(x_n^t - x_{n+1}^t)^2}, \tag{5.1}$$

where $a_{n+1}^{t+\Delta t}$: n+1$^{\text{th}}$ vehicle's acceleration at t+Δt
v_n^t : n^{th} vehicle's speed at t+Δt
x_n^t : n^{th} vehicle's position at t+Δt
$a = u_f$, (u_f = free flow speed).

Greenshields' model results in the following equations, establishing relationships among three fundamental traffic parameters: speed, density and traffic flow.

$$u = u_f\,(1- k / k_j)$$

$$q = u_f\, k\, (1- k / k_j)$$

$$k = k_j\,(1- u / u_f),$$

where, u: speed (km/hour)
u_f : free flow speed
k: density (vehicles/km)
k_j: jam density
q: flow (vehicle/hour).

It should be remarked that the model's appropriateness in traffic studies does not necessarily guarantee its goodness in representing aircraft behavior on the airfield. In fact, the parameters in generalized vehicle-following model should be calibrated with field data to warrant its use in airport studies. This is an open area of research for now.

If we assume that the Greenshield's model adequately represents the aircraft-following logic, then another approach for aircraft-following model can be devised. [Van Aerde, 1998] Using the Greenshields' basic speed-headway relationship, the method first determines the so called *desired speed* of the following aircraft at time t which is the desired to be reached at time $t+\Delta t$. Then the acceleration (or deceleration) is determined considering the current speed and other exogenous constraints related to the maximum permissible acceleration (or deceleration). The procedure is outlined below.

Step 1: Determine the following aircraft desired speed to be reached at time $t+\Delta t$ using Greenshields' basic speed-headway relationship.

$$v_{t+\Delta t}^{d} = v^{f}\,(1-\frac{H_j}{H_t})\,, \tag{5.2}$$

where $v_{t+\Delta t}^{d}$: following aircraft's desired speed at $(t+\Delta t)$
v^f : free flow speed
H_j : headway at jam density
H_t : headway between the leading vehicle and the following vehicle at time t.

Step 2: Obtain the following aircraft acceleration for the time interval from t to $t+\Delta t$ using the simple kinematic equation.

$$a_{n+1}^{t+\Delta t} = (v_{t+\Delta t}^{d} - v_t)/\Delta t\,;\ \text{if } a_{n+1}^{t+\Delta t} > a_{\max} \text{ then } a_{n+1}^{t+\Delta t} = a_{\max}, \tag{5.3}$$
$$\text{if } a_{n+1}^{t+\Delta t} < d_{\max} \text{ then } a_{n+1}^{t+\Delta t} = d_{\max}\ .$$

Here, $a_{n+1}^{t+\Delta t}$: following aircraft acceleration during the period from t to $t+\Delta t$

$v_{t+\Delta t}^{d}$: following aircraft desired speed at $(t+\Delta t)$

v_t : following aircraft's current speed at time t
Dt: the predefined time interval
a_{max}/d_{max}: maximum acceleration/deceleration.

In order to limit the kinematic behavior of the aircraft within a reasonable range, a non-uniform acceleration model is employed. This model assumes that the vehicle's maximum ability to accelerate (a_{max}) decreases linearly as a function of speed (see Figure 5.12). Another consideration for maximum deceleration (d_{max}) is based on passenger comfort.

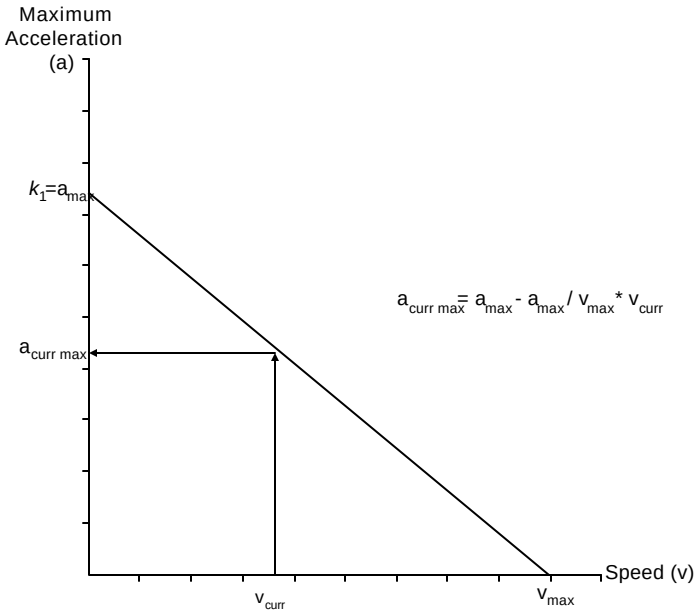


Figure 5.12 Speed-Acceleration Relationship.

The non-uniform acceleration model is considered to be reasonable because it adopts a thrust lapse rate with respect to aircraft speed. The same behavior has been adopted to describe the aircraft kinematic behavior on taxiways and taxilanes except that the values of a_{max} and v_{max} are adjusted accordingly to reflect lower speeds on these links. The equation for the non-uniform acceleration model is expressed as follows.

$$a\left(=\frac{dv}{dt}\right)=k_1-k_2 v \tag{5.4}$$

where, k_1, k_2: constants
v: speed
a: acceleration.

The equations for the speed-time relation can be deduced by separation of variables and integration of Equation (5.4).

$$\int_{v_0}^{v}\frac{dv}{(k_1-k_2 v)}=\int_0^t dt\,.$$

$$v=\frac{dx}{dt}=\frac{k_1}{k_2}(1-e^{-k_2 t})+v_0 e^{-k_2 t}\,. \tag{5.5}$$

The equation for the distance-time relationship is obtained in analogous way,

$$\int_{v_0}^{v} dv=\int_0^t\left(\frac{k_1}{k_2}(1-e^{-k_2 t})+v_0 e^{-k_2 t}\right)dt\,.$$

$$x=\frac{k_1}{k_2}t-\frac{k_1}{{k_2}^2}(1-e^{-k_2 t})+\frac{k_1}{k_2}(1-e^{-k_2 t})\,. \tag{5.6}$$

By substituting (5.5) into (5.6), we obtain the acceleration-time relationship.

$$a=\frac{dv}{dt}=(k_1-k_2 v_0)e^{-k_2 t}\,. \tag{5.7}$$

The resulting equations are summarized in Table 5.1. (In the case that $a_{\max}$ and $v_{\max}$ are given, Equation (5.5) is of the form $a=a_{\max}-kv$, where, $k = a_{\max} / v_{\max}$.) It should be remarked that the analytical solutions presented are only applicable to the leading vehicle. This is because these analytical solutions explain the behavior of a vehicle starting from an initial speed and accelerating as rapidly as possible in uninterrupted traffic flow. In fact, it is difficult to derive a closed-form solution to explain the following vehicle behavior. To overcome this shortcoming, the vehicle-following equations are implemented as a "continuous" simulation model using standard numerical integration techniques.

Table 5.1 Equations for the Non-Uniform Acceleration Model.

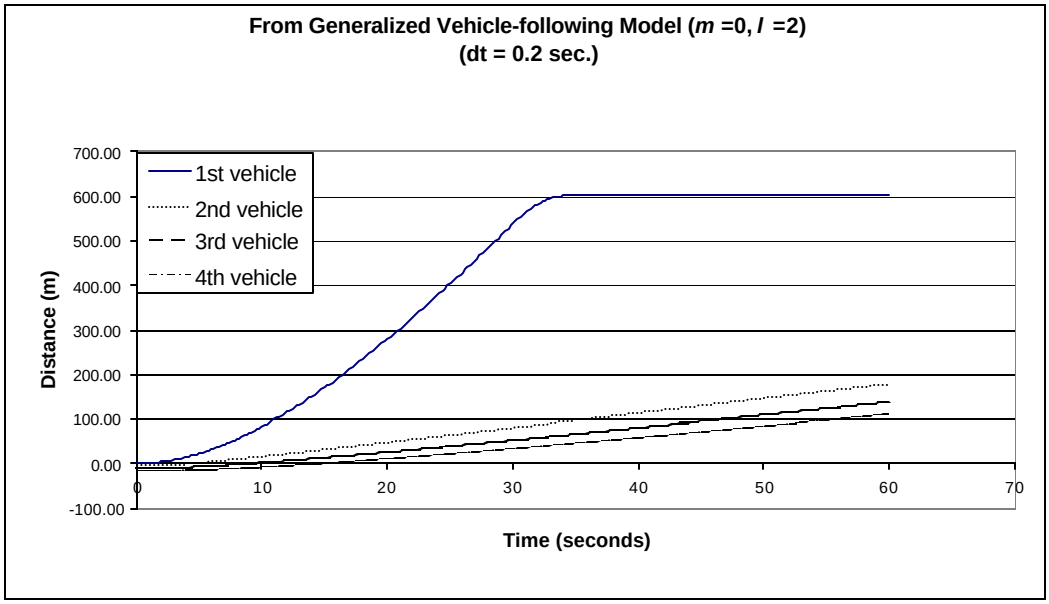
	Numerical Solution	Analytical Solution	
		not given a_{max} and v_{max}	given a_{max} and v_{max}
a_t =	$a_{max} - kv_{t-1}$	$(k_1 - k_2 v_0)e^{-k_2 t}$	$k(v_{max} - v_0)\exp(-kt)$ *1)
V_t =	$v_{t-1} + a_{t-1}\Delta t$	$\frac{k_1}{k_2}(1-e^{-k_2 t}) + v_0 e^{-k_2 t}$	$v_0 + (v_{max} - v_0)(1 - \exp(-kt))$
x_t =	$x_{t-1} + \frac{v_t + v_{t-1}}{2}\Delta t$	$\frac{k_1}{k_2}t - \frac{k_1}{k_2^2}(1-e^{-k_2 t}) + \frac{k_1}{k_2}(1-e^{-k_2 t})$	$x_0 + v_0 \cdot t + (v_{max} - v_0)(t - \exp(-kt)/k) - (v_{max} - v_0)/k$

*1) $k = a_{max} / v_{max}$

In order to compare the models resulting from Equations (5.1) and (5.3), consider the following simple example. At a taxiway-runway intersection, four aircraft are waiting for advisories from a local controller. Using the aircraft performance values shown below, we construct the resulting time-space diagrams according to the vehicle-following models described by Equations (5.1) and (5.3).

v_f = 60 kph
v_{max} = 120 kph
a_{max} = 2 m/s^2
$a_{emergency}$ = -.6g ≈ -6 m/s^2
H_j = 5 m.

The time-space diagram for the vehicle-following model described by Equation (5.1) is displayed in Figure 5.13 (a). This plot corresponds to a classical Greenshields' model (i.e., m=0 and l=2). Figure 5.13 (b) illustrates the vehicle-following behavior using Van Aerde's model. As seen in the graphs, there is a substantial difference in vehicle behavior. The two methods exhibit a disagreement in the headway parameter. The second vehicle in the generalized car-following model, in fact, does not seem to follow the first one. (The same outcome results even if dt is changed to as small a number as 0.02 sec.) The behavioral patterns observed persist even for small step sizes (dt = 0.02). Consequently, the modeling approach adopted in this research follows Equation (5.2) and (5.3). Figure 5.14 illustrates a flowchart showing the necessary steps to implement the aircraft-following models adopted in VTASM.



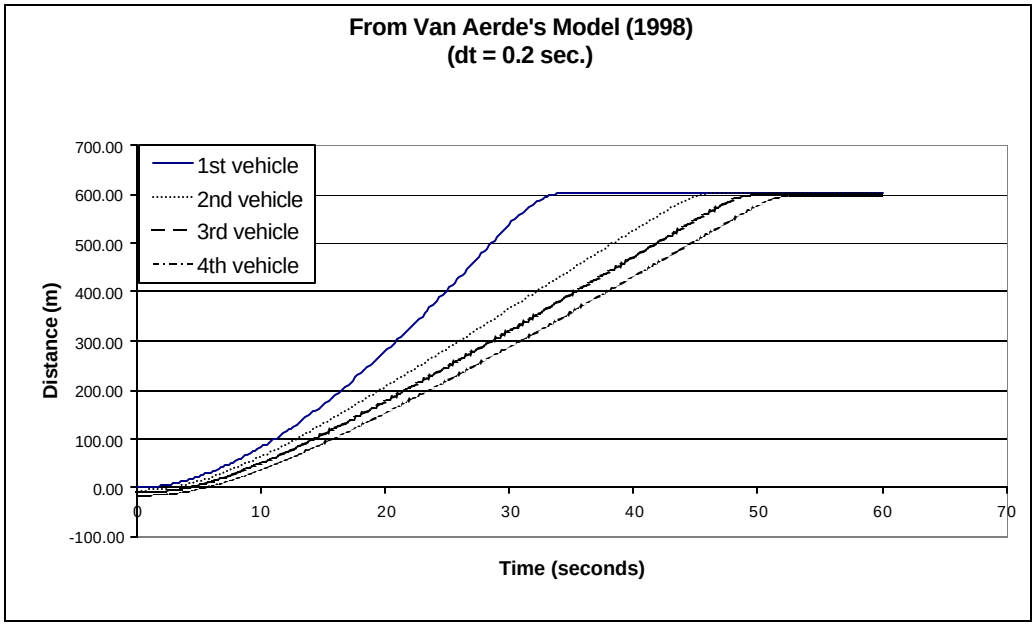


Figure 5.13 Comparison of Aircraft-following Models.

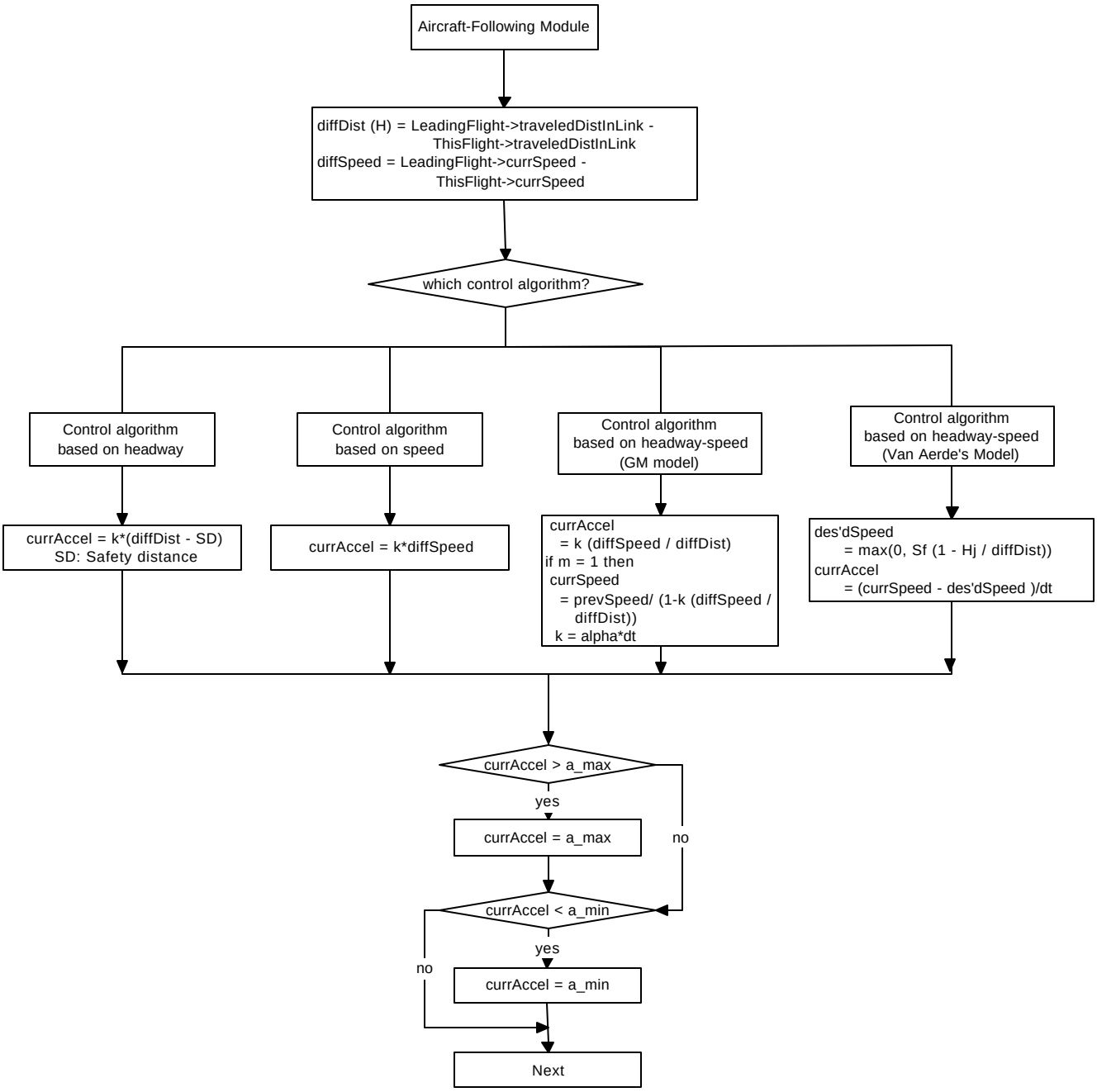


Figure 5.14 Flowchart of Aircraft Following Module.

5.3.2.1.2 An Algorithm for Conflict Detection and Resolution

A. Algorithm for Conflict Detection.

In the simulation model developed, explicit checks for potential ground collisions are implemented only for the first aircraft on a taxiway link crossing a check-point upstream of an intersection. All other following aircraft on the same link are not directly affected by any potential conflicts since they just follow the leading aircraft according to the aircraft-following logic explained in the previous section. For the first aircraft, the conflict detection method examines if there are any opposing aircraft on the adjacent links. One of following three circumstances could potentially result in a conflict between two aircraft. (See Figure 5.15.)

Case 1) The next taxiway link for aircraft 1 (current aircraft) is open and the next taxiway link for aircraft 2 (conflicting aircraft) is also open.

Case 2) The next taxiway link for aircraft 1 is blocked by aircraft 2 and the next taxiway link for aircraft 2 is open.

Case 3) The next taxiway links for both aircraft 1 and 2 are blocked by another aircraft.

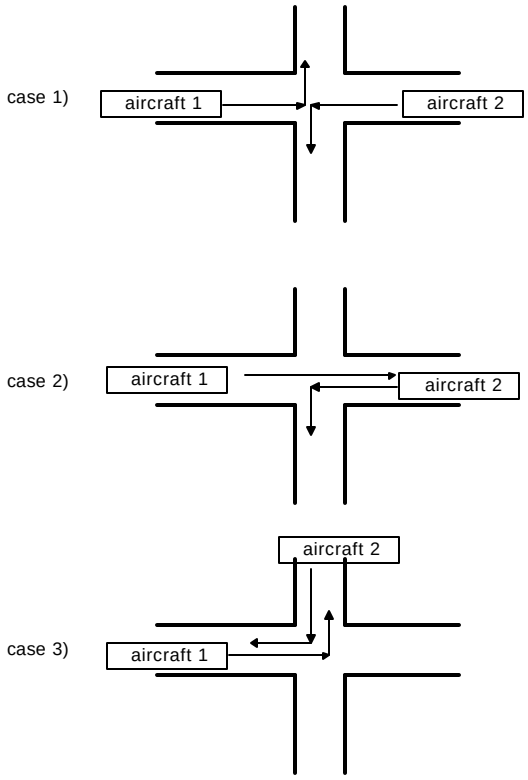


Figure 5.15 Cases of Potential Ground Conflicts.

For Case 2, there is no alternative for aircraft 1 except to wait until aircraft 2 crosses the intersection. In this case, the current aircraft needs to reduce its speed (or stop if necessary) allowing the other aircraft to pass the intersection without a substantial speed reduction. If Case 3 is triggered, there is no outlet for both aircraft (grid lock case). If Case 1 is invoked, the arrival times for two conflicting aircraft (aircraft 1 and 2, here) are estimated. If the expected arrival times of two aircraft are too close to secure a predefined minimum gap, the resolution routines are automatically invoked.

Consider the example shown in Figure 5.16. Assume that the current aircraft considered is "F and that the minimum gap for safe separation at the intersection is set to 15 seconds. Given that F1's current speed is 20 mi/hr (about 30 ft/sec) and the normal deceleration to stop is 3 ft/sec^2, F1 needs 10 seconds and 150 ft. to stop normally. Let us assume that F1 checks the collision risk at the point of 150 ft (i.e., 10 seconds in time) upstream from the intersection and that other aircraft, F2 and F3, are expected to enter the intersection at times, 30 and 15 seconds, respectively. In this hypothetical situation, unless a speed control adjustment is applied to some aircraft, the current aircraft, F1, will violate the required minimum separation gap with respect to aircraft F3. In this instance, a potential collision is detected and the ground conflict resolution logic is initiated at the same time.

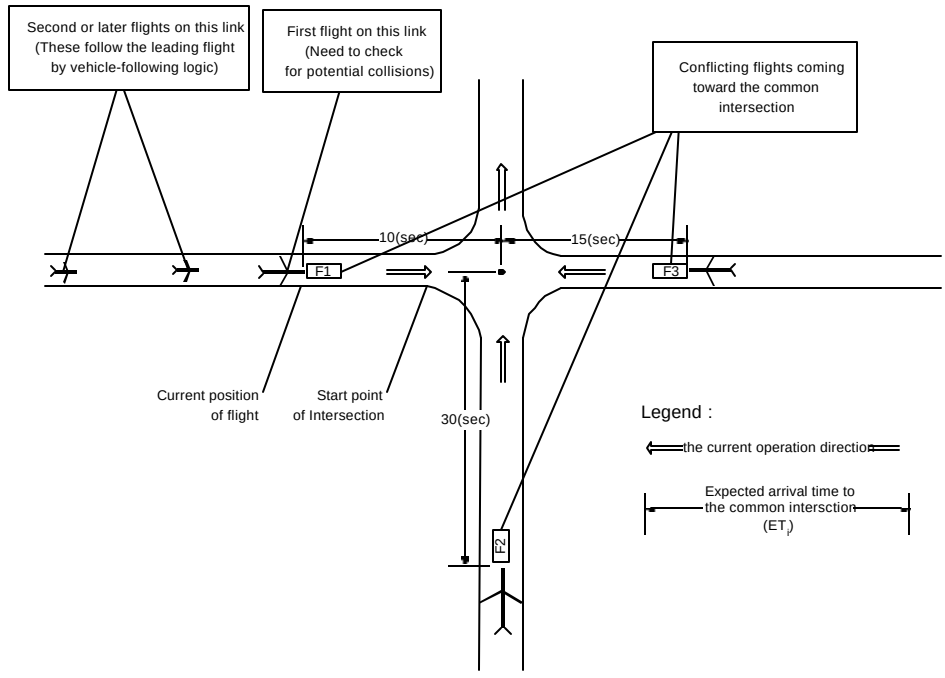


Figure 5.16 Concept of Conflict Detection.

B. Algorithm for Conflict Resolution

In the ground collision resolution logic, two issues are involved: 1) the flight priority and 2) the speed control logic. The controller can give a higher priority to certain flights over others with the intent of improving the overall efficiency of the taxiway (or runway) operations. The higher-priority flight has a right-of-way whenever it conflicts with another flight having a lower-priority. In other words, lower-priority flights should slow down or completely stop in order to secure a minimum gap at the intersection. In this case, the deceleration rate for the lower-priority flight can be estimated by using the speed-control logic. The following equations are involved in this speed control logic procedure.

$$EAT_{low}^{new} = EAT_{low}^{old} + \max[\text{minimum gap} - (EAT_{low}^{old} - EAT_{high}), 0] .$$

$$\frac{2 \cdot dist}{v_{low}^{current} - v_{low}^{intersection}} = EAT_{low}^{new} .$$

$$v_{low}^{intersection} = v_{low}^{current} - \frac{2 \cdot dist}{EAT_{low}^{new}} .$$

Hence, $$decel_{low} = \frac{v_{low}^{intersection} - v_{low}^{current}}{EAT_{low}^{new}} = -\frac{2 \cdot dist}{EAT_{low}^{new^2}} . \quad (5.8)$$

Here EAT_{low}^{new} (EAT_{low}^{old}): the low priority flight's new (old) expected arrival time at the intersection,

EAT_{high} : the high priority flight's expected arrival time at the intersection,

$v_{high}^{current}$: the low priority flight's present speed,

$v_{high}^{intersection}$: the low priority flight's speed at the intersection,

$dist$: distance from the current position to the intersection,

$decel_{high}$: the low priority flight's deceleration rate.

Figure 5.17 illustrates the basic speed relationships related to the conflict resolution logic. Here, two areas encompassed by a dotted line and a solid line produce the distance from the current position to the intersection. In the foregoing example shown in Figure 5.15, if two conflicting aircraft (F1and F3) have the same priority, the current flight (F1) will have a right-of-way based on the first-in-first-out (FIFO) rule. The deceleration rate for flight F3 will be determined by Equation (5.8). The complete logical procedures for collision detection and resolution that are implemented in the simulation model are Figure 5.18.

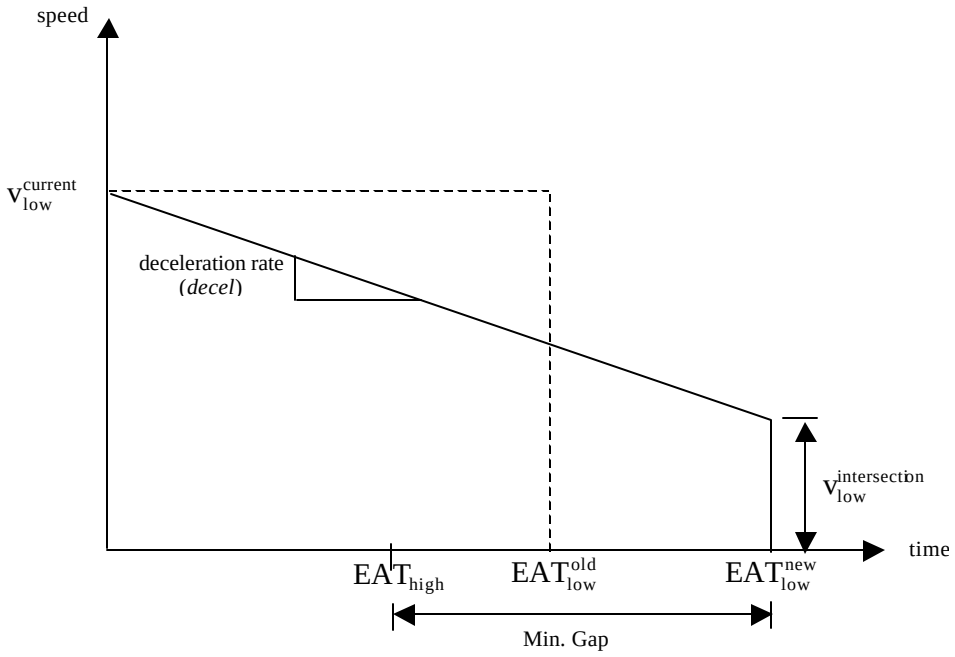


Figure 5.17 Calculation of the Lower-priority Flight's Deceleration Rate.

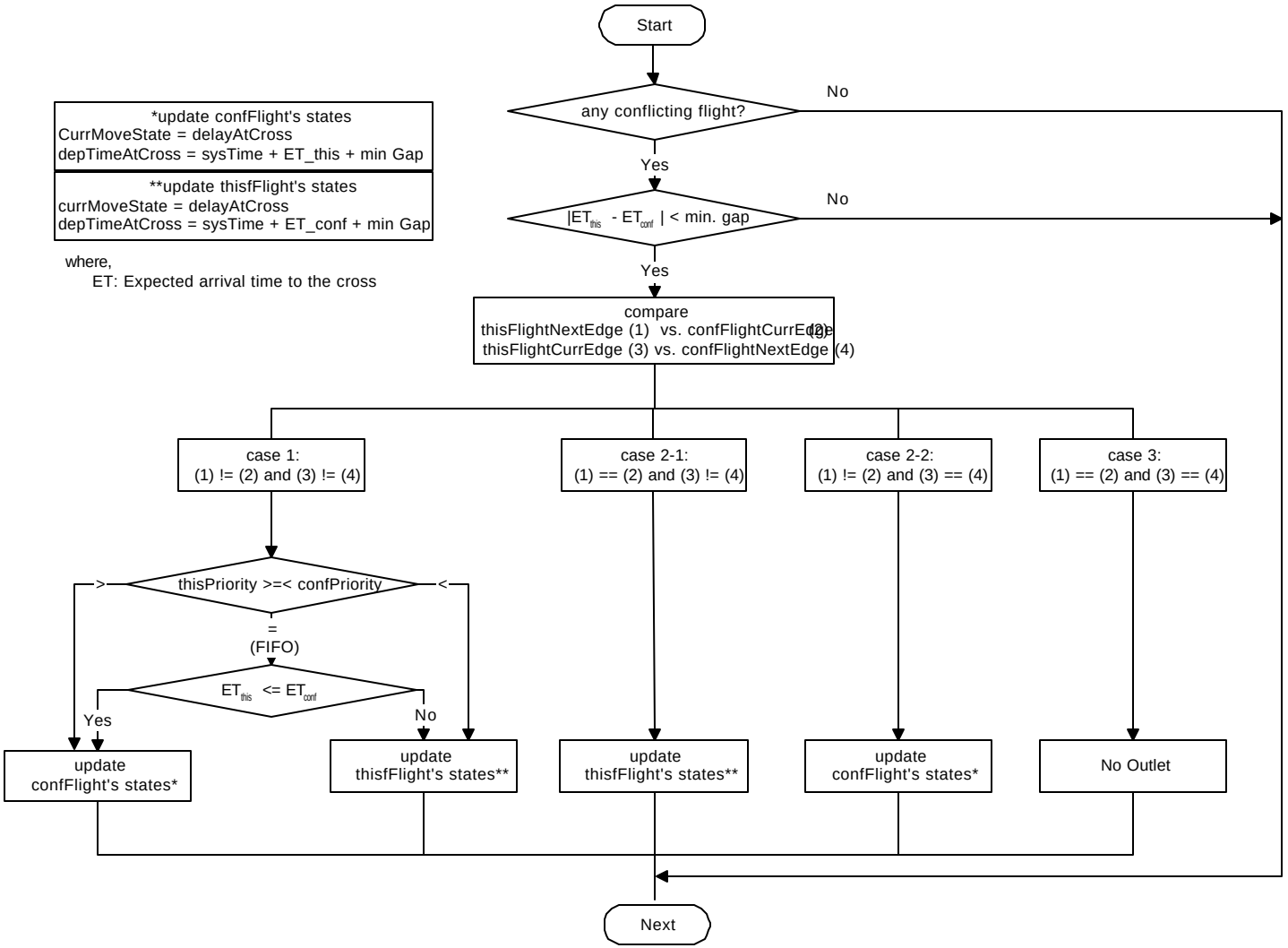


Figure 5.18 Flowchart for the Conflict Detection and Resolution Module

5.3.2.1.3 Algorithms for the Takeoff and Landing Procedures

A. Algorithms for the Takeoff Procedure

Once a departing flight reaches the runway and obtains takeoff clearance from the (local) controller, it initiates the takeoff procedure using its maximum acceleration potential. The takeoff procedure can be divided into two phases: the takeoff rolling phase and the lift-off phase [Trani, et al., 1993]. As the aircraft speed reaches lift-off speed, the aircraft becomes airborne and wheels off from the ground. These two phases of the takeoff procedure are depicted in Figure 5.19. To simplify its implementation, the lift-off phase is assumed to be characterized by a constant speed after lift-off. The simulation model keeps track of the departing flight until it crosses the runway threshold. In general, the time from the lift-off point to the runway threshold is relatively small as the aircraft travels at a high speed.

There are two important parameters related to the takeoff rolling analysis: the takeoff rolling distance and the takeoff rolling time. These values are estimated in the simulation model using a aircraft equation of motion that is integrated forward in time.

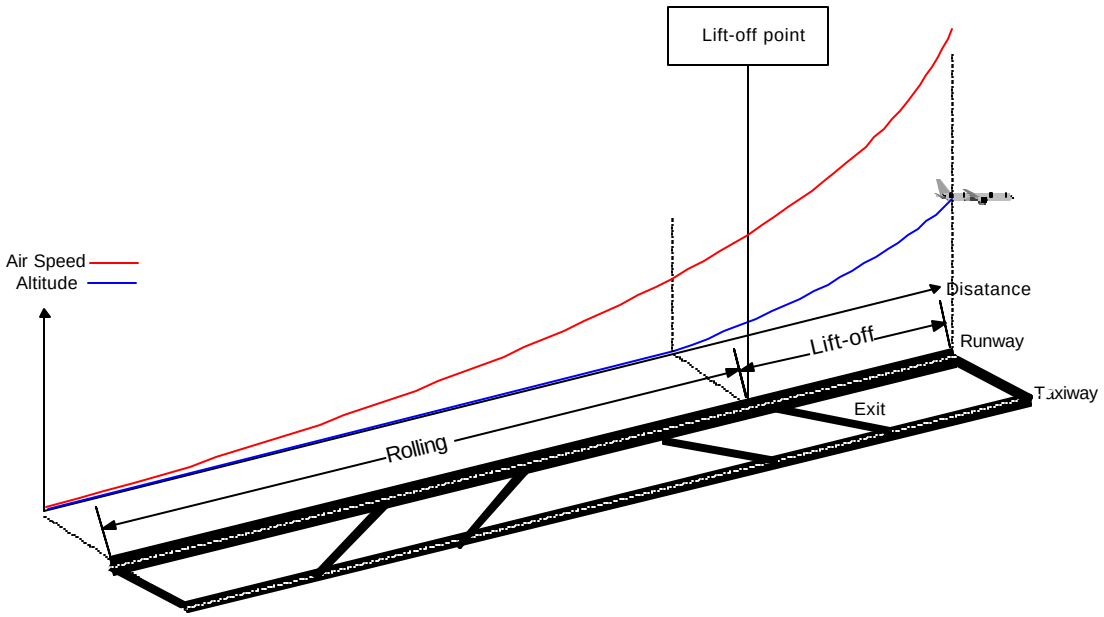


Figure 5.19 Takeoff Procedure.

The steps for the takeoff roll analysis are illustrated in Figure 5.20. Following traditional aerodynamic performance estimation calculations, the first step is to convert all atmospheric conditions to an equivalent international atmospheric status. This is because the performance of an aircraft engine is affected by the atmosphere around airport. Once the corrected engine thrust has been estimated in Step 2, the takeoff roll distance is computed.

The takeoff rolling time estimation requires some knowledge of two aircraft aerodynamic parameters: the drag coefficient (C_d) and the lift coefficient (C_l). Usually, these parameters are not made public by aircraft manufacturers. Consequently, approximations are needed to estimate takeoff rolling times via alternative methods. A simple alternative method adopted in this model is an adaptation of takeoff roll algorithms used in the FAA Integrated Noise Model (INM 6.0).

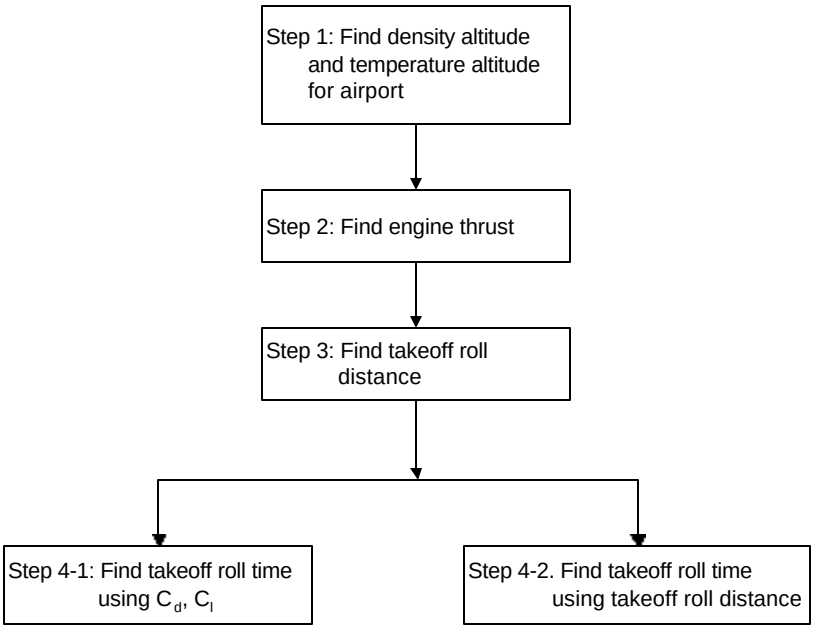


Figure 5.20 Steps for the Takeoff Roll Analysis.

Step 1: *Find density altitude and temperature altitude*. Using ISA (the International Standard Atmosphere) properties as shown in Table 5.2, the temperature at a given altitude can be calculated using the standard temperature lapse rate [Lan, 1981]:

$$T = T_0 + l(h - h_0),$$

where T: temperature at altitude h (°F),
T_0: temperature (°F) and altitude at mean sea level (h_0=0 ft),

h: airport elevation (ft),
h_0: elevation of mean sea level (h_0=0 ft),
l: atmospheric temperature lapse rate (°F/ft).

The pressure and density variations with altitude can be calculated using the following standard thermodynamic equations:

$$q = \frac{T}{T_0} = 1 + \frac{l}{T_0} h = 1 - 6.875 \times 10^{-6} h,$$

$$d = \frac{P}{P_0} = \left(\frac{T}{T_0} \right)^{-\frac{1}{lR}} = \left(1 + \frac{l}{T_0} h \right)^{-\frac{1}{lR}} = q^{5.2561},$$

$$s = \frac{r}{r_0} = \frac{P}{P_0} \times \frac{T_0}{T} = \frac{d}{q} = \left(\frac{T}{T_0} \right)^{-\frac{1}{lR}-1} = q^{4.2561},$$

where,

q : temperature ratio at airport's current altitude (which is the airport elevation here),
h: airport elevation (ft),
d : pressure ratio at current aircraft altitude,
P: atmospheric pressure at altitude h (in lbs/ft^2 or N/m^2),
P_0: atmospheric pressure at sea level (in lbs/ft^2 or N/m^2),
l: atmospheric temperature lapse rate (in °F/ft or °C/m),
R: thermal gas constant (in ft/°R or m/°K),
T_0': standard sea level temperature, absolute value (in °R or °K),
l: atmospheric temperature lapse rate (in°F/ft or °C/m),
s : air density ratio at aircraft's current altitude (by employing the ratio version of the ideal gas law),
r : air density at altitude h,
r_0: air density at sea level standard condition.

To illustrate the use of the equations, suppose that a standard altimeter indicates an altitude of an altitude of 15,000 ft when the ambient air temperature is 35 °F. Let us accordingly calculate the density altitude and the temperature altitude.

1) Density altitude:

At h = 15,000 ft, the standard temperature is 5.5 °F (=59-0.00356616(15000)). Hence, the atmosphere is not standard. Since the altimeter is a pressure gauge, it will read the correct pressure. The correct pressure at that altitude would be 1194 psf (= 2116.2 × (temperature ratio) ^ 5.2561 = 2116.2 × ((5.5+459.7) / 518.7) ^ 5.2561)).

Table 5.2 Conversion Factors and Properties of International Standard Atmosphere.

		SI unit	UK unit	Conversion Factor
ISA	Gravity constant (g_0)	9.806 m/sec^2	32.17 ft/sec^2	
	Pressure (p_0)	1.013×10^5 N/m^2	2116.2 lb/ft^2	1 lb/ft^2 = 47.880 N/m^2 (=Pa)
	Temperature (T_0)	15 °C	59 °F	°F = 9/5(°C+32)
	Air density (ρ_0)	1.225kg/m^3	0.002377 slug/ft^3	1 slug = 14.59kg
etc.	Absolute temperature (T)	°K = °C + 273.15	°R = °F + 459.7	
	Atmospheric temperature lapse rate (l)		-0.00356616 °F/ft	
	Thermal gas constant	29.26 m/°K	53.35 ft/°R	
	Force	Newton (kgm/sec^2)	lb-force	1 lb-force = 4.448 Newton
	Weight	kg	pound	1 pound = 0.453 kg
	Length	m	ft	1 ft = 0.3048 m
	Speed	1 knot = 0.514 m/sec		

(Source: Lan and Roskman (1981) at http://www.ex.ac.uk/cimt/dictunit/dictunit.htm#length).

$$\rho = \frac{p}{gRT} = \frac{1194}{32.17(53.35)(35+459.7)} = 0.001406 \text{ (slug/ft}^3\text{)}.$$

$$s = \frac{\rho}{\rho_0} = \frac{0.001406}{0.002377} = 0.5915 .$$

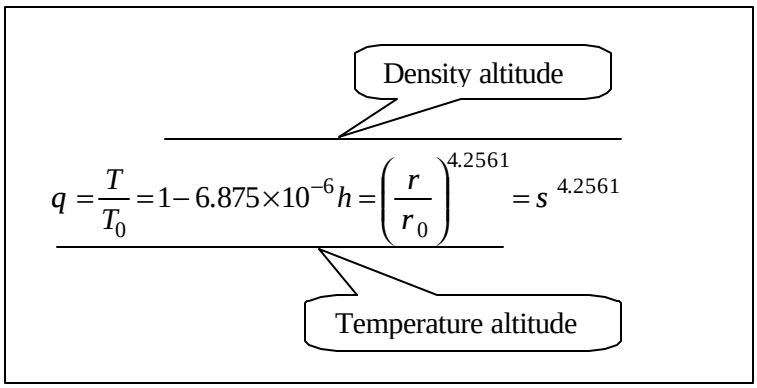
$$q = 1 - 6.875\times10^{-6} h_d = s^{1/4.2561} = 0.88393 .$$

Therefore, $h_d = \frac{1-0.88393}{6.875\times10^{-6}} = 16{,}883$ (ft).

2) Temperature altitude:

$$q = \frac{T}{T_0} = \frac{35+459.7}{518.7} = 0.95373 = 1 - 6.875\times10^{-6} h_t .$$

Therefore, $h_t = \frac{1-0.95373}{6.875\times10^{-6}} = 6{,}730$ (ft).



Step 2: *Compute the net thrust per engine for jets.* The non-standard atmosphere net thrust per engine for jets, F_n, is a function of calibrated airspeed v, and density altitude h_d, is calculated using the SAE-AIR-1845 thrust equation developed by Society of Automotive Engineers. The non-ISA thrust equation accounts for thrust-reducing effects of hot temperatures at high altitudes [INM User's Guide, 1996].

$$F^{jet}_n(v,h_d) = d(h_d)[E_p + F_p \times v + G_{1p} \times h_d + G_{2p} \times h_d^2 + H_p \times T_s(h_d)],$$

where, F^{jet}_n: non-ISA thrust per engine for jets (lb),
$d(h_d)$: ISA pressure ratio at aircraft's density altitude h_d,
v: calibrated airspeed (kt),
h_d: the aircraft's density altitude which is the elevation of the airfield in our problem (ft),
T_s: ISA temperature at the aircraft's density altitude (°C),
$T_s = (5/9)\,(59 - 0.003566 \times h_d - 32)$,
$E_p, F_p, G_{1p}, G_{2p}, H_p$: the engine dependent regression constants which depend on the jet's power setting state (max-takeoff or max-climb) from INM database.

$$F^{jet}_{corrected} = F^{jet}_n / d.$$

The corrected net thrust per engine for props, $F^{prop}_{corrected}$, is given by

$$F^{prop}_{net\text{-}standrad} = 325.87\ h_p P_p / V_t\ ,$$

where, $F_{net\text{-}standard}$: the standard atmosphere net thrust per engine for props,
h_p: propeller efficiency, which depends on the power-setting state,
P_p: net power engine (hp) for sea-level standard day,
V_t: true airspeed (kt) = $V_c \cdot s^{-1/2}$.

$$F^{prop}_{corrected} = F^{prop}_{net\text{-}standrad} / d.$$

Step 3: *Find takeoff roll distance.* For the analysis of takeoff rolling distance, the initial and final values of aircraft altitude are given as the airport elevation. The horizontal distance traveled on the ground is calculated after initial and final values of speed and thrust are calculated. The procedure follows SAE-AIR-1845.

The takeoff rolling distance (S_g) is given by,

$$S_g = B_f \cdot q \cdot \frac{(mg/d)^2}{N \cdot F_{net-corrected}}$$

where , S_g : takeoff roll distance (ft),

B_f : an airplane specific coefficient for a given flap deflection combination,
m: aircraft mass (lb),
g: gravity acceleration (= 32.2 ft/sec^2),
q : temperature ratio at aircraft altitude,
d : pressure ratio at aircraft altitude,
N: the number of engines per aircraft,
$F_{\text{net-corrected}}$: corrected net thrust per engine (lb) at takeoff rotation.

Typical velocities for the calculation of thrust are,

v_1 = 16 knots,

$v_2 = C_f(mg)^{1/2}$,

$v_2' = v_2/(s)^{1/2}$,

where, v_1: calibrated speed at the beginning of the takeoff roll maneuver (i.e., at the break release point) predefined as 8.2 m/sec or 16 knots,
v_2 : calibrated speed at the end of the takeoff roll maneuver (i.e., at lift-off point),
$C_{f,}$: coefficient which depends on the flap setting (from the INM database),
v_2 : true air speed at the end of the takeoff roll maneuver,
s : air density ratio at aircraft altitude.

Step 4-1: *Find takeoff roll time using C_l and C_d.* Forces related to the takeoff roll distance are [Trani, et al., 1993],

$$L = \frac{1}{2} r v^2 S C_l ,$$

$$D = \frac{1}{2} r v^2 S C_d ,$$

$$TE = f(v, r) ,$$

$$F_f = (mg\cos f - L) f_{roll} ,$$

where, L = lifting force (lb),
D = drag force (lb),
TE: corrected net tractive effort (lb),
F_f: friction force (lb),
r: air density (slugs/ft^3),
S: the aircraft reference area (ft^2),
C_l: lift coefficient which is dependent on the flap setting (non-dim),
C_d: drag coefficient which is dependent on the flap setting (non-dim),
f_{roll}: rolling friction coefficient (non-dim) normally 0.02~0.03,

f : the runway inclination angle with respect to the horizontal (degree).

The forces in the *x* (horizontal direction) produce

$$ma_x = TE(v, r) - D - (mg\cos f - L) f_{roll} - mg\sin f \ .$$

Hence,

$$a_x = \frac{1}{m}\left(TE(v, r) - D - (mg\cos f - L) f_{roll} - mg\sin f\right).$$

Assuming a linear tractive effort lapse rate function of speed, the following expression can be derived,

$$TE(v) = T_1 - \left(\frac{T_1 - T_2}{v_2}\right)v,$$

where, TE: tractive force at speed v,
T_1, T_2: tractive force at brake release point, and lift-off point,
v_2: aircraft speed lift-off point.

Note that runways are near flat by regulation (i.e., less than 2% effective gradients (f) for general aviation (GA) runways and 1.5% for transport runways) and thus for the takeoff analysis, the angle f can be neglected in most practical applications. Neglecting the effect of grade on the acceleration of the vehicle, we can estimate the following accelerations for two reference conditions: 1) at brake release ($a_{x|br}$), and 2) at the lift-off point ($a_{x|lo}$).

$$a_x = \frac{1}{m}\left(T_1 - \left(\frac{T_1 - T_2}{v_2}\right)v - \frac{1}{2}\rho v^2 S(C_d - C_l f_{roll}) - mgf_{roll}\right),$$

$$a_{x|br} = \frac{1}{m}\left(T_1 - mgf_{roll}\right),$$

$$a_{x|lo} = \frac{1}{m}\left(T_2 - \frac{1}{2}\rho v_2^2 S(C_d - C_l f_{roll}) - mgf_{roll}\right).$$

If we further assume that the acceleration varies inversely with the speed range from the brake release point to the lift-off point, the simplified acceleration at speed v can be evaluated as follows:

$$a_x\left(=\frac{dv}{dt}\right)=a_{x|br}-\left(\frac{a_{x|br}-a_{x|lo}}{v_2}\right)v$$

$$=\frac{1}{m}(T_1-mgf_{roll})-\left(\frac{\frac{1}{m}(T_1-T_2)+\frac{1}{m}\left(\frac{1}{2}\,rv_2^2S(C_D-C_l f_{roll})-mgf_{roll}\right)}{v_2}\right)v$$

$$=k_1-k_2v\,,$$

where, $k_1=\frac{1}{m}(T_1-mgf_{roll})$,

$$k_1=\left(\frac{\frac{1}{m}(T_1-T_2)+\frac{1}{m}\left(\frac{1}{2}\,rv_2^2S(C_d-C_l f_{roll})-mgf_{roll}\right)}{v_2}\right).$$

By integrating once from the brake release speed (v_1) to the lift-off speed (v_2), the takeoff roll time, t_{roll}, is estimated as

$$t_{roll}=-\frac{1}{k_2}\left(\frac{k_1-k_2v_2}{k_1-k_2v_1}\right).$$

Step 4-2: *Find takeoff roll time without using C_l and C_d.* Here, we introduce an alternative method to approximate takeoff roll distances without any information on C_l and C_d. The estimation is based on the assumption of a non-uniform acceleration model discussed earlier in this section. The resulting equations for the acceleration, speed and distance-time relationships are summarized in Table 2.9, and are depicted graphically in Figure 5.21. Here, Equations (5.9)-(5.11) are used again.

$$a=\frac{dv}{dt}=k_1-k_2v \tag{5.9}$$

$$v=\frac{dx}{dt}=\frac{k_1}{k_2}(1-e^{-k_2t})+v_0e^{-k_2t}\,. \tag{5.10}$$

$$x=\frac{k_1}{k_2}t-\frac{k_1}{k_2^{\,2}}(1-e^{-k_2t})+\frac{v_1}{k_2}(1-e^{-k_2t})\,. \tag{5.11}$$

By substitution of (5.10) into (5.11), we obtain

$$a = \frac{dv}{dt} = (k_1 - k_2 v_0) e^{-k_2 t}. \tag{5.12}$$

The next step is to find parameters k_1 and k_2 which characterize the Equations (5.10), (5.11) and (5.12). Using the SAE-AIR-1845 model, assuming the speed at the beginning of the takeoff roll, v_1 (= v_{br}), to be 8.2 m/sec or 16 knots, and letting

v_2 (= v_{lo}) : speed at the point of lift-off,
a_1 (= a_{br}) : the acceleration at the beginning of takeoff roll,
a_2 : the acceleration at the beginning of lift-off,

the equation for the time to reach a certain speed v_2 is derived from (5.10) as

$$t_2 = -\frac{1}{k_2} \ln\left(\frac{k_1 - k_2 v_2}{k_1 - k_2 v_1}\right). \tag{5.13}$$

By substitution of (5.13) into (5.11), we obtain

$$x = \frac{k_1}{k_2}\left(-\frac{1}{k_2}\right)\ln\left(\frac{k_1 - k_2 v_2}{k_1 - k_2 v_1}\right) - \frac{k_1}{k_2^2}\left(1 - e^{\ln\left(\frac{k_1 - k_2 v_2}{k_1 - k_2 v_1}\right)}\right) + \frac{v_1}{k_2^2}\left(1 - e^{\ln\left(\frac{k_1 - k_2 v_2}{k_1 - k_2 v_1}\right)}\right). \tag{5.14}$$

Since the conditions at the beginning of takeoff roll are known, $v = v_1$, $a = a_1 = a_{br}$, from (5.9), k_1 can be expressed as k_2 according to

$$k_1 = a_1 + k_2 v_1. \tag{5.15}$$

Substituting (5.15) into (5.11), the equation for the distance as a function of k_2 can be obtained as

$$\begin{aligned} x = {} & \frac{a_1 + k_2 v_1}{k_2}\left(-\frac{1}{k_2}\right)\ln\left(\frac{(a_1 + k_2 v_1) - k_2 v_2}{(a_1 + k_2 v_1) - k_2 v_1}\right) \\ & - \frac{a_1 + k_2 v_1}{k_2^2}\left(1 - e^{\ln\left(\frac{(a_1 + k_2 v_1) - k_2 v_2}{(a_1 + k_2 v_1) - k_2 v_1}\right)}\right) + \frac{v_1}{k_2^2}\left(1 - e^{\ln\left(\frac{(a_1 + k_2 v_1) - k_2 v_2}{(a_1 + k_2 v_1) - k_2 v_1}\right)}\right) \end{aligned} \tag{5.16}$$

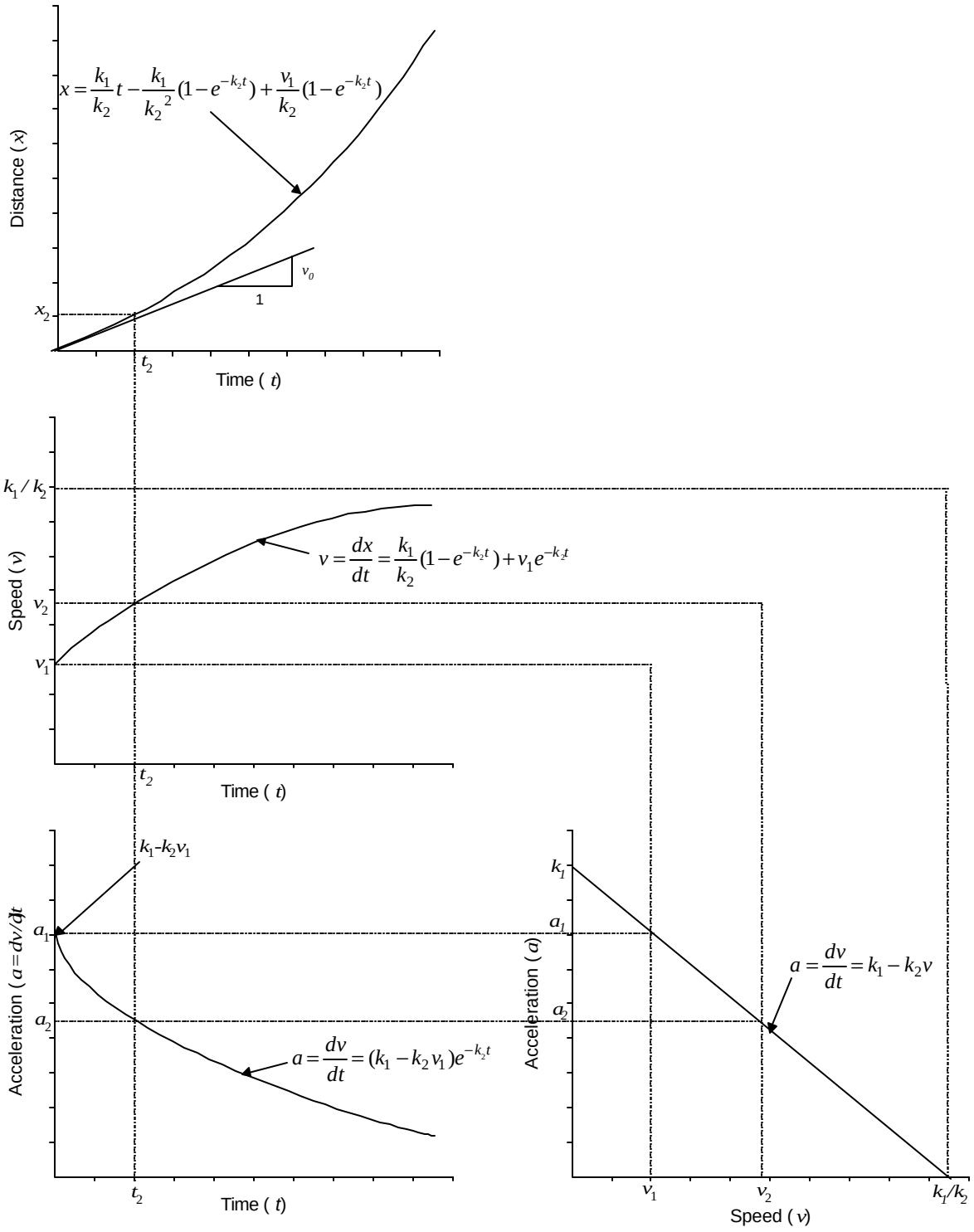


Figure 5.21 Non-Uniform Acceleration Model [Drew, 1968].

Given v_1, v_2, a_1 and the takeoff roll distance, x (=S_g), there is only one unknown variable, k_2 in Equation (5.16). Because k_2 cannot be readily represented in a closed-form, numerical methods can be used to obtain k_2. Once k_2 is available, k_1 is calculated using (5.16).

One important factor that should not be overlooked in the takeoff roll analysis is that two variables, takeoff roll distance and time, are rather stochastic than deterministic. In other words, these variables may also be explained with certain types of statistical distribution models extracted from observed data.

In general, the stochastic behavior observed during the takeoff roll has been addressed by federal regulation authorities by imposing a correction factor to distance calculated via analytical methods. This correction factor increases the runway length by 15% to account for pilot deviations in the takeoff roll. This criteria does not apply in this model since VTASM assumes the runway length to be sufficient for landing and departing operations.

B. Algorithms for the Landing Procedure

The landing procedure can be divided into four phases: flare, free-rolling, braking, and coasting phases. (See Figure 5.22). The flare segment begins at the moment when the aircraft crosses the runway threshold and ends when the main landing gear touches down on the runway. Pilots tend to maintain a conservative margin over the stalling speed (v_{stall}), which can be calculated by the following equation. The initial speed of this phase, v_{fl}, is empirically known to be about 1.25 times v_{stall}.

$$\text{Stall speed } (v_{stall}) = \sqrt{\frac{2mg}{\rho C_{l\max} A_w}}, \tag{5.17}$$

where m: the aircraft mass (kg)

g: gravity acceleration

ρ : standard atmosphere air density (kg/m^3)

$C_{l\max}$: maximum landing lift coefficient

A_w: the aircraft wing area (m^2).

The flare distance (s_{air}) is measured from the runway threshold to the touchdown point and is expressed mathematically as [Trani et al., 1993],

$$s_{air} = d_l + d_c + DRL \tag{5.18}$$

$$d_l = \frac{h_{th}}{g} \tag{5.19}$$

$$d_c = \frac{v_{fl}^2 \cdot g}{2g(n_{fl} - 1)} \tag{5.20}$$

$$DRL = DL \times RL, \tag{5.21}$$

where, s_{air}: flare distance
d_l: linear descending distance from the runway threshold to the touchdown aiming point
d_c: circular-arc flare maneuver distance for transiting to a touchdown attitude with a minimum sink rate and
DRL: the correction distance that is influenced by the runway length.

Also,

v_{fl}: threshold crossing airspeed (m/sec or ft/sec)
v_{td} : touchdown speed (m/sec or ft/sec)
s_{air}: touchdown distance (m or ft)
h_{th}: threshold crossing height (m or ft)
g : tangent value of the descent flight path angle (degree),
g: acceleration of gravity (m/sec^2 or ft/sec^2)
n_{fl}: the flare load factor
DL: correction distance factor (meter for every 100 m (328 ft) of runway length, valid for 2100m < RL < 2800m)
RL: runway length
t_{air}: duration time in flare (m or ft)

$$v_{stall} = \sqrt{\frac{2mg}{rCL_{\max} A_w}}$$

m: the aircraft mass
g: gravity acceleration
r : the standard atmosphere air density
CL_{max}: maximum landing lift coefficient
A_w: the aircraft wing area.

The free-rolling phase starts at the point where the main gear touches down and ends when thrust-reverse and/or braking are applied. It has been observed that the duration of the free-rolling phase is about 1-3 seconds with an average deceleration rate of 0.70 m/sec^2 [Trani et al., 1993].

The braking phase is initiated from the ending point of the free-rolling phase and completes at the moment when the aircraft decelerates to the so-called "decision speed" (about 35 m/s for heavy aircraft), when the pilots decide which exit will be used. According to the aircraft type, different deceleration rates are applied in the braking phase.

Once the aircraft passes the decision point, it begins to coast to the runway exit. During the coasting phase, the pilot controls the speed based on the current speed and distance between the current point and the selected exit. To capture this dynamic behavior, a second-order feedback control system is used in the model implementation.

The speed-distance profile among landing phases is sketched in Figure 5.23. The equations for each phase are summarized in Table 5.3 along with some statistical parameters which are

observed to fit a normal distribution model quite well. Table 5.4 shows the landing the roll statistics for various aircraft types observed at various airports [Kim et al., 1996].

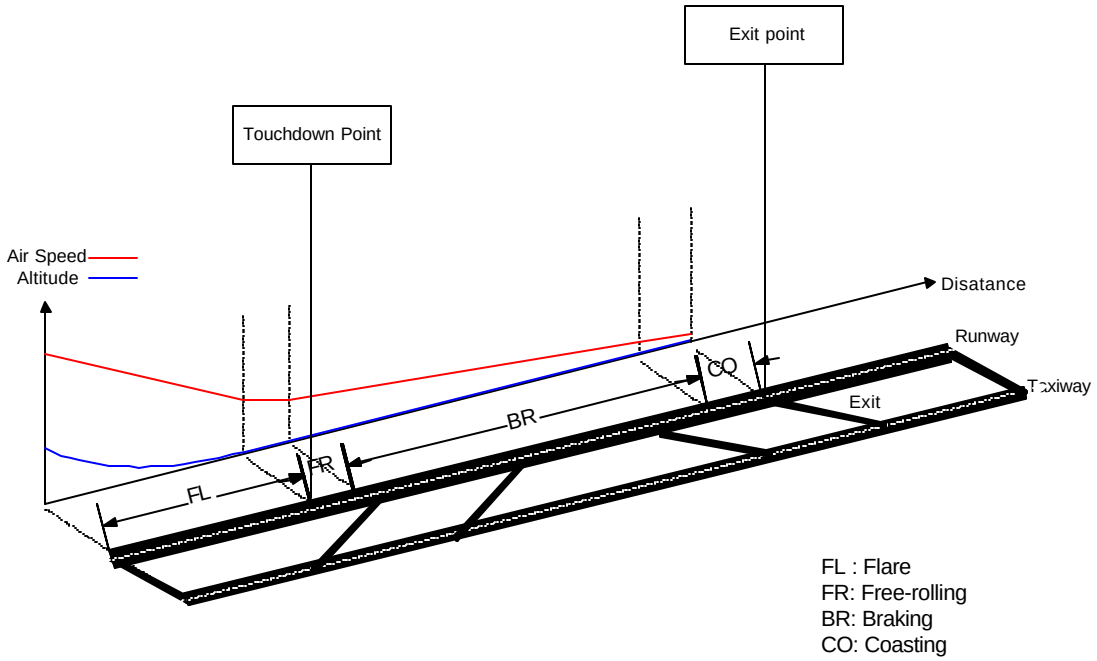


Figure 5.22 Four Phases in the Landing Procedure.

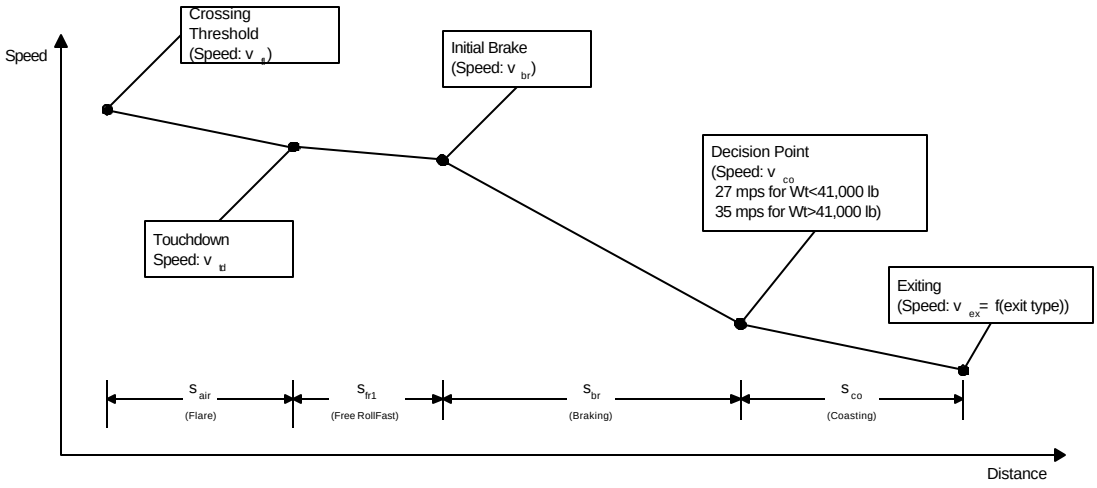


Figure 5.23 Speed-Distance Relationship for the Landing Procedure.

Table 5.3 Equations describing the Aircraft Landing Phases.

Phase	Initial Speed	Ending Speed	Distance	Duration	Acceleration	Empirical values
Flare [1]	v_{fl}	v_{td}	$s_{fl} = \frac{h_{th}}{g} + \frac{v_{fl}^2 \cdot g}{2g(n_{fl}-1)} + \Delta L \cdot RL$	$t_{fl} = \frac{s_{fl}}{(v_{fl}+v_{td})/2}$	$a_{fl} = \frac{(v_{td}-v_{fl})}{t_{fl}}$	$\gamma \cong 2.5\sim3$ degree (or ~N(2.75, 0.08^2)) $h_{th} \cong 15.2\ m$ (= $50ft$) (or ~N(15.2, 3^2)) n_{fl} = 1.1~1.3 $DL \cong$ +25m (=82 ft)/100m runway length $v_{fl} \cong 0.95\ v_{ap} \cong 1.25\ v_{stall}$ (or ~ $N(\bar{v}_{fl}, 0.06 \cdot \bar{v}_{fl}^2)$) $v_{td} \cong 0.95\ v_{fl}$ (or $v_{fl} - 3.2$)
Free Rolling [2]	v_{td}	$v_{br} = v_{td} - a_{fr}t_{fr}$	$s_{fr} = v_{td}t_{fr} - \frac{a_{fr}t_{fr}^2}{2}$	t_{fr}	a_f	t_{fr} = 2~3 seconds $a_{fr} \cong 0.7$ m/sec^2
Braking [3]	v_{br}	$v_{co} = v_{br} - a_{br}t_{br}$	$s_{br} = v_{br}t_{br} - \frac{a_{br}t_{br}^2}{2}$	t_{br}	a_{br}	t_{br} = time to decelerate to the speed of 27mps for wt < 41000 lb 35mps for wt > 41000 lb $a_{br} = f$(aircraft type)[5] B727~N(2.19, 0.416^2) B737~N(2.25, 0.471^2) B757~N(2.01, 0.478^2) DC-9~N(2.03, 0.414^2) MD-80~N(2.05, 0.387^2)
Coasting [4]	v_{co}	v_{ex}	s_{co} (by 2nd-order feedback control)	t_{co} (by 2nd-order feedback control)	a_{co}	v_{ex} = f(exit type)

1) v_{fl}: threshold crossing airspeed (m/s or ft/s)
v_{td} : touchdown speed (m/s or ft/s)
s_{air}: touchdown distance (m or ft)
h_{th}: threshold crossing height (m or ft)
g : tangent value of the descent flight path angle (degree)
g: acceleration of gravity (m/s^2 or ft/s^2)
n_{fl}: the flare load factor (DIM)

δ(RL): correction distance (meter for every 100 m (328 ft) of runway length, valid for 2100m < RL < 2800m).
t_{air}: Duration time in flaring out (m or ft)

$$v_{stall} = \sqrt{\frac{2mg}{\rho C L_{\max} A_w}}$$

m: the aircraft mass
g: gravity acceleration
ρ : air density
CL_{max}: maximum landing lift coefficient
A_w: the aircraft wing area

2) v_{br}: Initial braking speed (m/s or ft/s)
a_{fr}: Average free-rolling deceleration (m/s^2 or ft/s^2)
t_{fr}: Free-rolling time (seconds)
3) v_{co}: Initial coasting speed (m/s or ft/s)
a_{br}: Average braking deceleration (m/sec^2 or ft/s^2)
t_{br}: Braking time (seconds)
4) v_{ex}: Exit speed (m/sec or ft/s)
5) Kim, et al., 1996.

Table 5.4 Landing-roll Statistics [Kim et al., 1996].

Airport (Runway)	Runway Length	Grade[1)] (%)	Aircraft Type	No. of Obs.	Flaring speed (m/s)		Touchdown distance (m)		Braking deceleration (m/s^2)	
					Mean	S.D.	Mean	S.D.	Mean	S.D.
DCA (R36)	2094 m (6869 ft)	0	B-727	72	66.62	3.03	455.0	132.1	2.26	0.382
			B-737	36	65.77	3.99	399.2	80.0	2.30	0.422
			B-757	26	65.30	5.78	424.9	97.7	2.14	0.675
			DC-9	36	65.02	3.54	434.9	105.8	2.08	0.397
			MD-80	51	68.29	4.51	434.3	94.1	2.14	0.428
CLT (R23)	2286 m (7500 ft)	-0.5	B-727	13	68.18	3.16	546.9	169.8	1.83	0.511
			B-737	34	66.08	3.57	400.0	77.4	2.21	0.573
			B-757	4	61.55	2.11	489.6	139.7	1.62	0.231
			DC-9	8	67.34	3.46	425.2	79.6	2.08	0.56
			MD-80	7	66.60	2.55	550.6	188.3	1.81	0.381
ATL (R8L)	2742 m (9000 ft)	-0.3	B-727	13	70.87	3.87	621.7	164.2	2.11	0.423
			B-737	12	68.74	4.34	603.3	75.9	2.08	0.497
			B-757	10	65.28	5.29	699.9	115.4	1.79	0.337
			DC-9	13	68.85	3.9	594.0	137.9	1.83	0.341
			MD-80	28	68.57	4.97	569.7	124.6	1.90	0.302

1) Deceleration rate decreases by 0.01 m/sec^2 (0.033 ft/sec^2) per 0.1% of grade change on runway.

5.3.2.2 Data Structures

Data structures play an important role in designing an efficient computer code because they govern the organization of the model information, and thereby constitute the basis for good algorithms. Two types of data structures are mainly used in implementing our simulation model: lists and queues.

A "list" is a finite, ordered sequence of data items known as elements ("ordered" in the definition means that each element has a position in the list). There are two implementations of a list: a static array-based list, and a dynamic linked list using pointers. In an array-based list, the size of the array should be fixed before the array is created. On the other hand, the linked list is dynamic in the sense that it allocates memory for new list elements as needed. There are advantages in the dynamic linked list such as added flexibility in programming. However, the linked list needs extra space to keep a pointer that indicates the next element of the list (in singly-linked list case). In terms of computational cost, the array-based list is faster in accessing the i^{th} element and appending an element to the tail of the array. Operations to remove and insert an element are relatively expensive. As a rule of thumb, linked lists are a better choice when working with vectors whose sizes are unknown or which vary widely. Array-based lists are generally more space efficient when the user knows in advance the size of the list [Shaffer, 1997]. Computational efficiencies of the two lists are compared in Table 5.5.

Table 5.5 Comparison of List Implementations.

		Array-based list	Linked list
advantage		no wasted space for an individual element	Need space for the objects actually on the list
cost	access i^{th} element	$\Theta(1)$ [a)]	$\Theta(i)$
	append an element	$\Theta(1)$	$\Theta(1)$
	insert i^{th} element	$\Theta(i)$	$\Theta(1)$
	remove i^{th} element	$\Theta(i)$	$\Theta(1)$

a) Θ (big-theta) indicates that the upper-bound, O(big-oh), and the lower bound, Ω (big-omega), are the same.

The array-based list is used in the simulation model to store information about nodes and links. This is efficient because the numbers of nodes and links at an airport are known in advance and can be assumed to remain unchanged during the period of simulation. This is similar to the database for the aircraft characteristics because the number of aircraft types is also limited and

fixed. Figures 5.29 and 5.30 illustrate the applications of array-based lists to the node data and aircraft data, respectively.

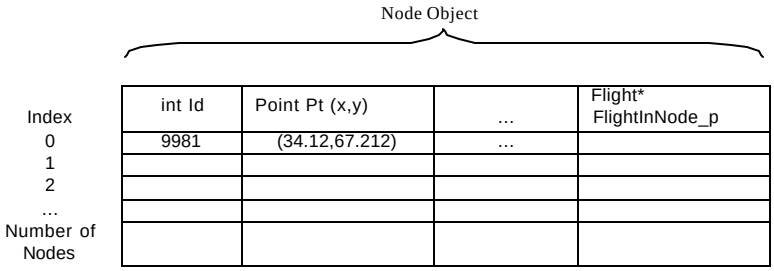


Figure 5.24 Array-based List for Node Data.

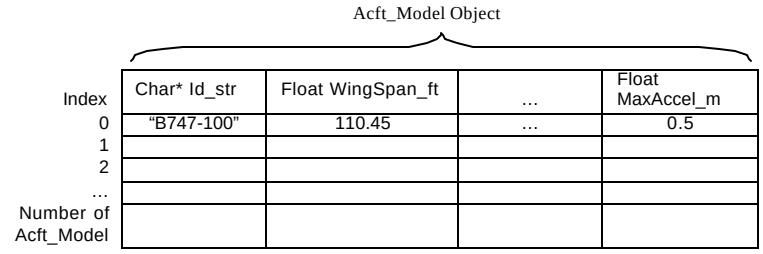


Figure 5.25 Array-based List for Aircraft Model Data.

In contrast, the number of flights on the flight schedule dynamically changes according to the day, or time of day. The taxiing path between the gate and the runway for a flight is also flexible in the sense that the number of links defining taxiing paths is not fixed. In these cases, a linked list, particularly a singly-linked list, is adopted to maintain the information regarding the flight schedules and taxiing paths. Figures 5.31 and 5.32 show describe the implementation of linked lists for these two variable size vectors.

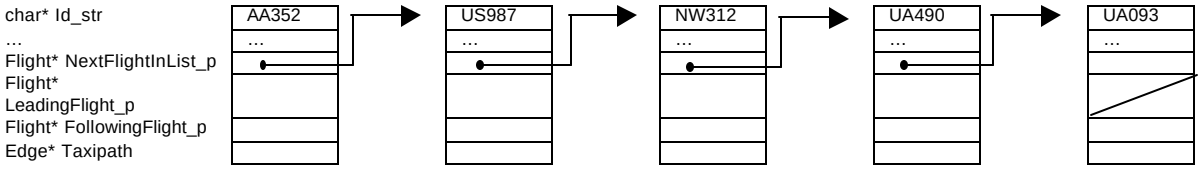


Figure 5.26 Singly-Linked List for Edge Data.

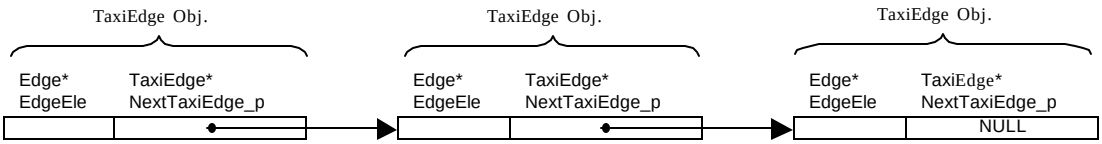


Figure 5.27 Singly-Linked List for Taxiing Path Data.

The adjacency list, which is commonly applied in graph theory, is used for representing the networks within the simulation model. (For more details, see Figure 2.22.) In practice, an array of a singly-linked list having $|n|$ items is applied to the adjacency list, where $|n|$ is the number of nodes. A sample network and an array for a linked list are shown in Figure 5.28.

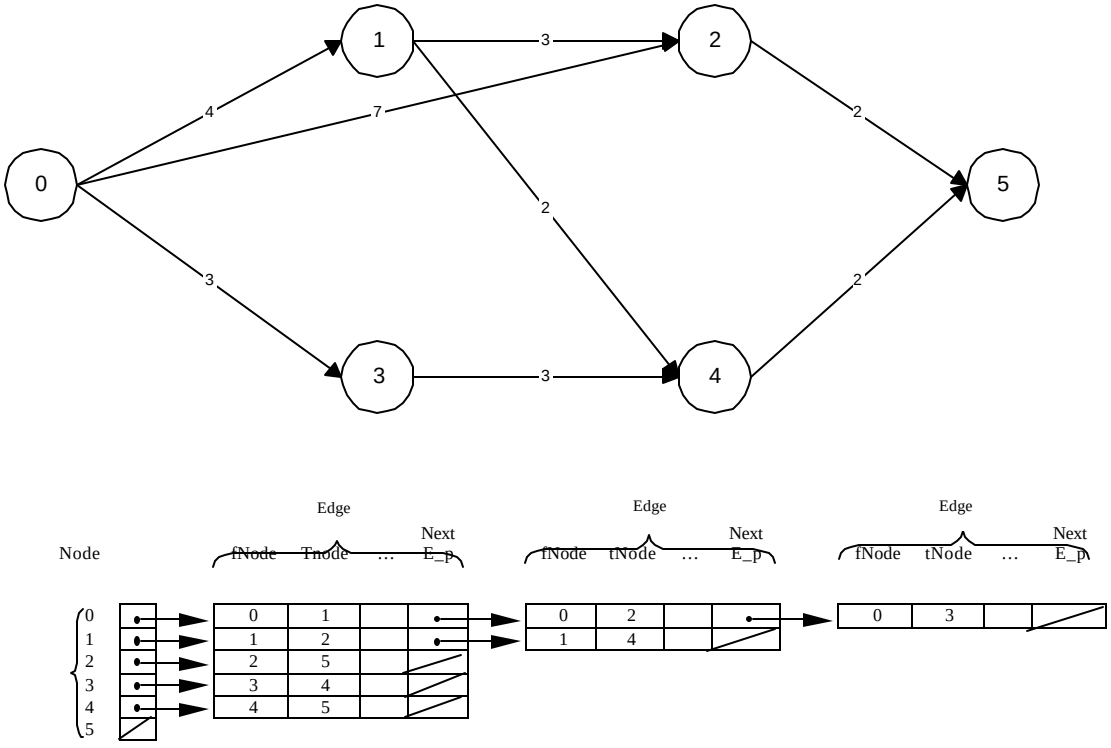


Figure 5.28 Sample Network and Array of Singly-Linked List for the Sample Network.

A FIFO queue (See Figure 3. 8) is a form of a restricted list, in which an element may only be entered at the back and removed from the front of the list. The service line at a bank is a typical example of a queuing system. The aircraft in a link can be also described by a queuing system. An aircraft entering into a new taxiway link is stored at the end of the queue.

As reviewed in the previous chapter (see Table 3.6), the sorted queue and the (output-restricted) double-ended queue are used in the implementation of the static shortest path algorithm and the time-dependent shortest path algorithm respectively. Table 5.6 summarizes the resulting data structures used in the implementation of the simulation model.

Table 5.6 Summary of Data Structures Used in the Simulation Model.

Data structure type		Data
List	array-based list	Node, Link
	linked list	Flight schedule, Taxiing path
Array of linked list		Airport Network
Queue	sorted queue	Dijkstra algorithm
	FIFO queue	Aircraft on the link
	output-restricted double-ended queue	Time-dependent shortest path algorithm

5.3.2.3 Flowcharts

The last step in the object model in OMT is to write pseudo-codes for algorithms or other related methods. Instead of pseudo codes, our approach is to use flowcharts showing the details of the coding process graphically. Figure 5.29 depicts the procedural flows for the complete simulation model.

The initialization procedure is illustrated in Figure 5.30. Basic objects such as the network graph, flights and controllers are generated and initialized after all related data is read from the input file. (See Appendix A for the list of input data.) A controller object calculates the shortest path between all nodes defining the airport network. The allocation of flights to each controller is also done during the initialization procedure. Some of the important variables associated with flights and air traffic controllers and their initial states are summarized in Table 5.7. Communication and movements are two types of activities involved in the aircraft flight behavior. The efficient organization of these activities centers around two event times named “nextCommEventTime”

and "nextMoveEventTime" which are created inside the flight objects (more detailed member variables and functions are included in Appendix C).

The main part of the simulation model is executed in such a way that the states of all entities i.e., flights, ground controller(s), local controller(s), in the system are updated every time interval until the simulation time ends. Two types of loops are involved in this process: an outer loop where the system clock proceeds by a time increment (Δt) until the simulation ends, and an inner loop where the states of all flights in the system are checked and updated successively.

Table 5.7 The Initial States of the Simulation Model Variables.

object	variable	initial state
controller	currState	standby
	nextEventTime	simulationDuration
flight	currCommState	readyToCommunicate
	nextCommEventTime	simulationDuration
	currMoveState	parking/onFinal*
	nextMoveEventTime	scheduled time
	position	gate/runway threshold
	speed	0.0/final approach speed
	acceleration	0.0
	needToComm	false
	collisionChecked	false
	permission	undecided

* State for arrival/departure.

Inside the inner loop, a check is made for each flight's time clock to determine the movement. ock is initially set to the time given in the flight schedule. When the system clock advances and passes the flight's scheduled time, the flight executes the two major activities related to communication and movement within the outer loop. Otherwise, all the processes inside the outer loop are just skipped.

Unlike the flight movement which is checked continuously, the communication activities are treated as discrete events. This is because a communication event is scheduled only when it is necessary. Communication states in both controller and flight objects change in a discrete fashion. The module named "checkNeedToComm" and illustrated in Figure 5.31 checks if a flight requires communication with either the local or the ground controller. If a flight attempts to communicate, two state variables, "needToComm" and "nextCommEventTime" are set to "true" and the current system time, respectively. The communication module (shown in Figure 5.3) initiates the communication events and changes the flight movement state if it is permitted to move.

In Figure 5.33, a communication process is depicted from the flight's point of view. Here, the state variable called "nextCommEventTime" is updated according to the time duration required for the current communication activity. On the other hand, the controller's communication states change according to the corresponding flight's current communication state as shown in Figure 5.34.

It should be noticed that a function called "judge" mimics the controller's decision process on whether a flight request is accepted. This function changes the controller's state via "judgingCommand". If the controller decision is to accept a flight to move and end communication, the flight's "nextCommEventTime" is set to the simulation duration so that the flight can jump the communication procedure until it needs to communicate again. Usually, the taxiing clearances from the local controller are provided with the taxiing route from the runway exit to the gate.

The states of flight movement such as speed, acceleration, position, etc., are continuously evaluated after a flight enters the system. The main concern in the movement logic is to decide how much the flight would accelerate (or decelerate) in the next time interval. Unless a flight is either stopped, parked at a gate, or waiting on a runway, the flight's dynamic behavior is decided by its own control logic depending on its current movement state. For example, if an arriving flight is in the coasting phase on the runway, its acceleration for the next time interval is determined by the second-order feedback control system discussed in the previous section.

Acceleration (or deceleration) during taxiing is a little more complicated because it might depend on a leading aircraft, if any, on potential collisions, and on the remaining distance to the destination for taxiing, etc. The detailed processes to decide the acceleration for a taxiing flight are summarized in Figure 5.36, and the flowchart for the conflict detection and resolution algorithm is previously shown in Figure 5.18.

The last procedure inside the inner loop is to update the flight kinematic states to reflect the current changes according to the new acceleration value selected. This is done by the "update" module shown in Figure 5.37. In the case that a flight enters a new link, the flight information on its leading and following flight as well as the link information are updated by "enqueue" and

Once the complete procedure for a flight has been executed, a check is made to decide if this flight is the last one on the list. If it is the last one, the inner loop is completed, and both ground and local controllers start to check if there is any flight awaiting controller's contact by looking at their flight progress strips. If there is any flight and the controller are both in standby states, contact is made by the controller. (See Figure 5.9 for details.) Once all controllers finish an

appropriate action, the simulation time advances by a time increment to commence another iteration of the outer loop (see Figure 5.29).

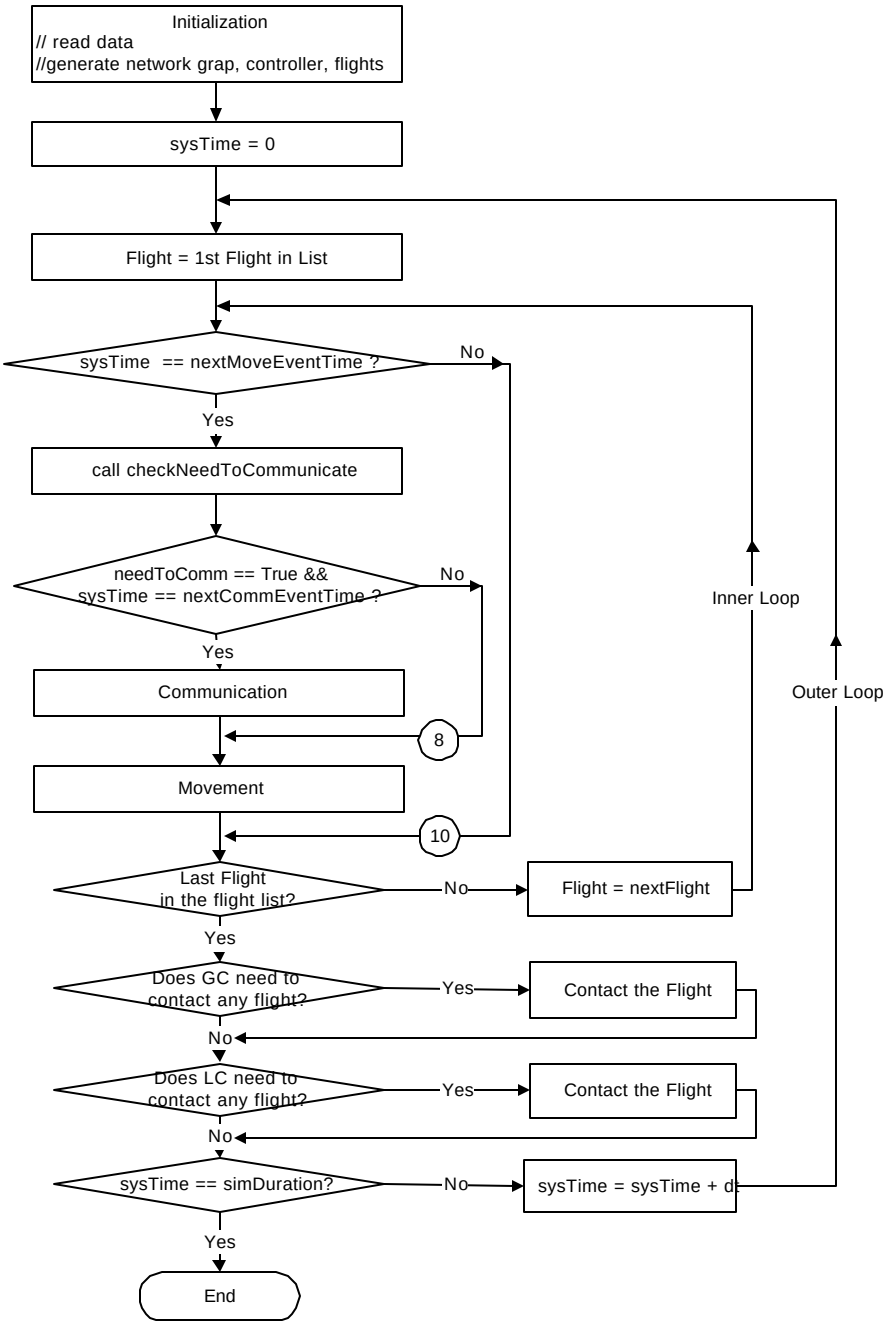


Figure 5.29 Flowchart for the Overall Simulation Model Process.

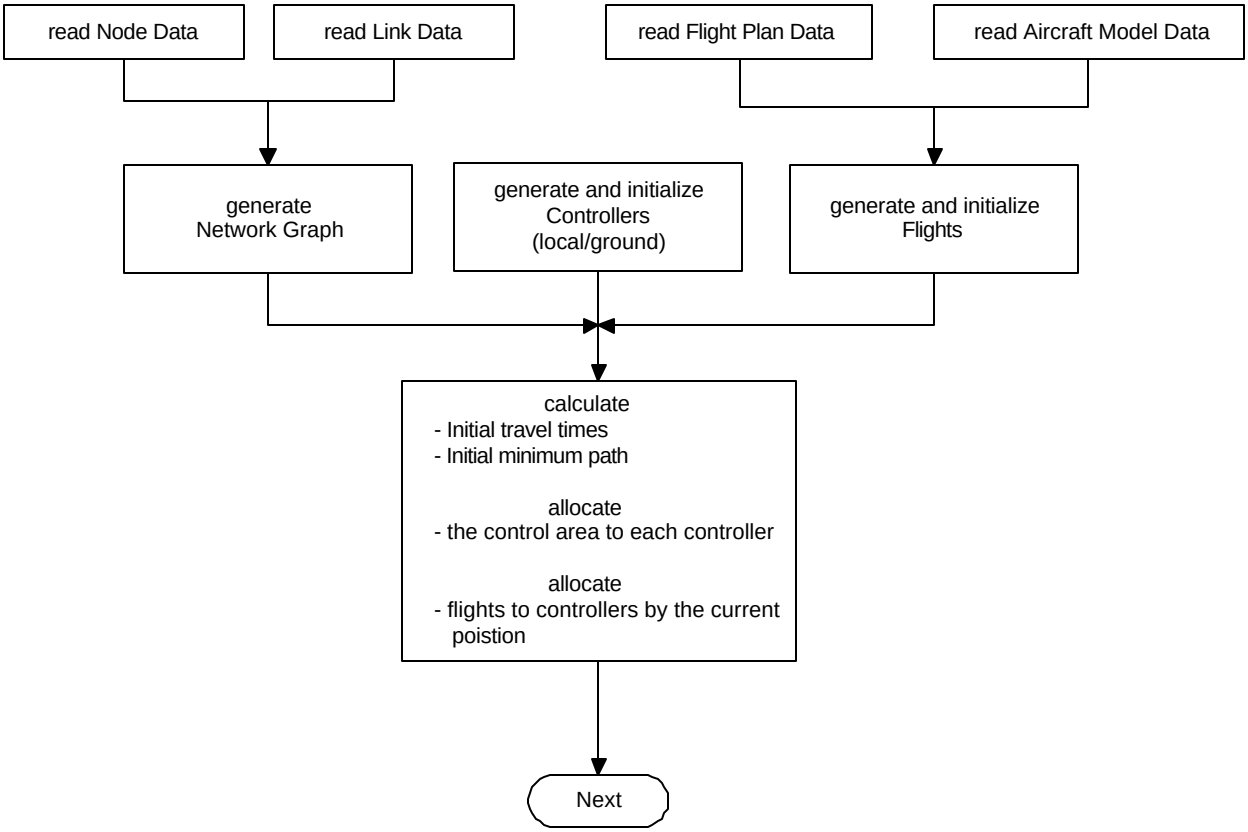


Figure 5.30 Flowchart for the Initialization Step.

a

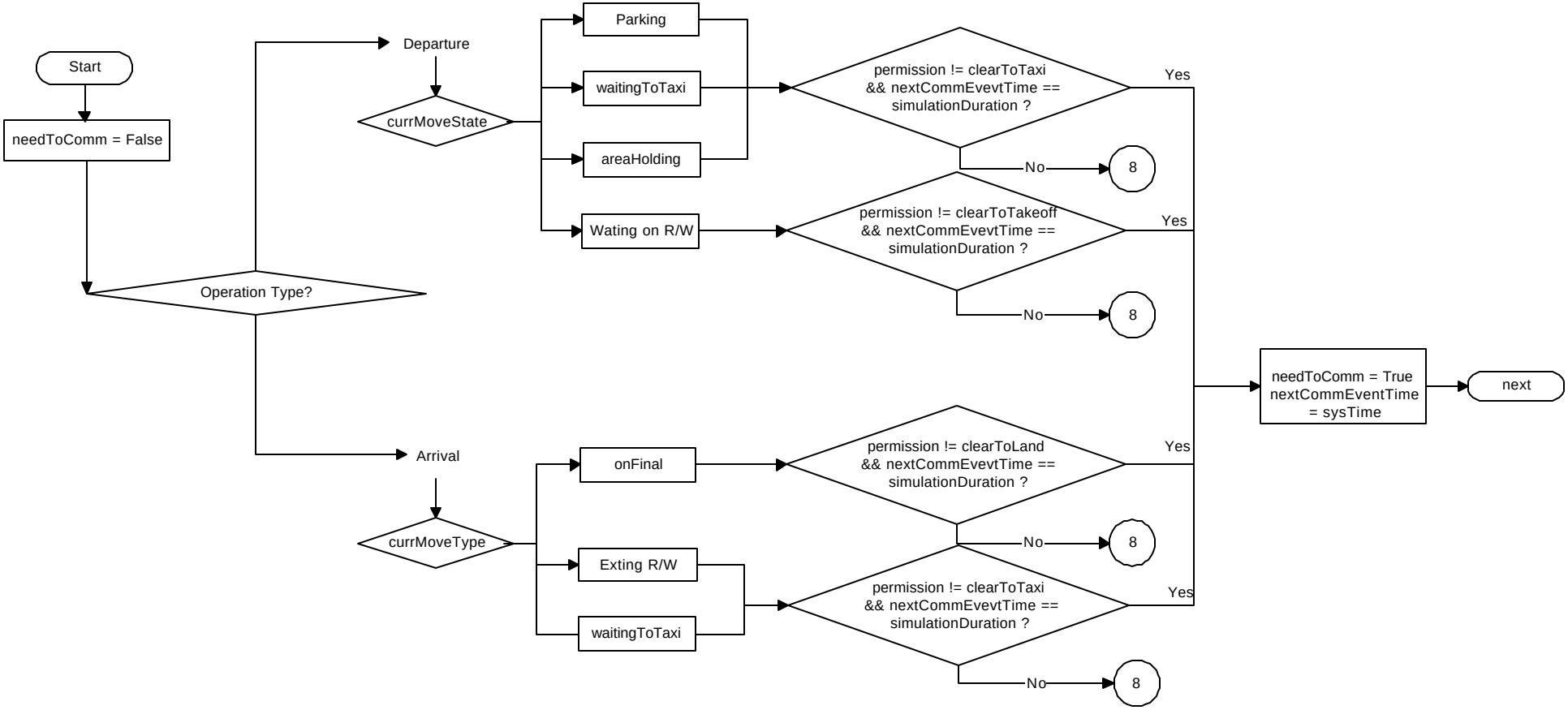


Figure 5.31 Flowchart for Performing Communication Checks.

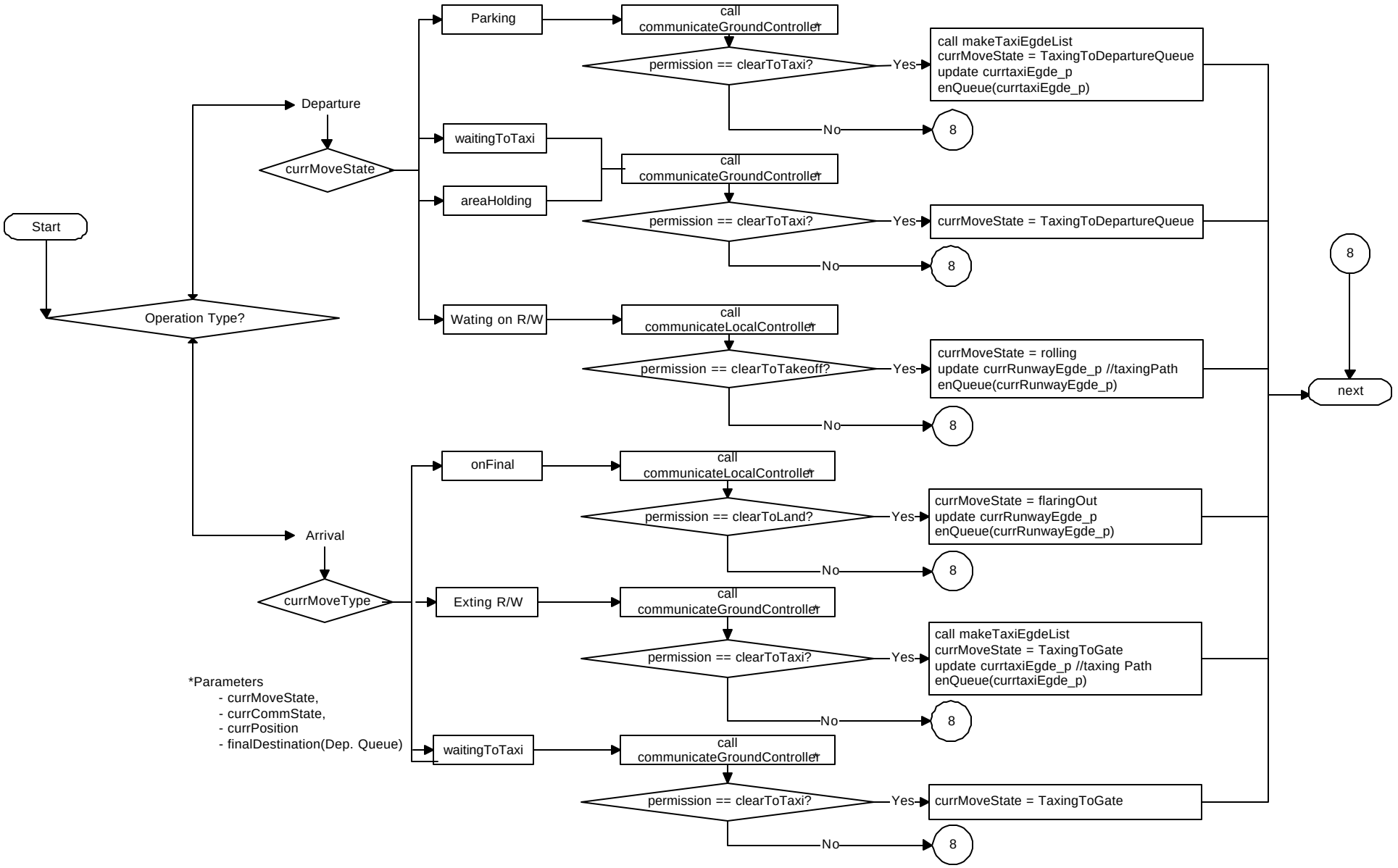


Figure 5.32 Flowchart for the Communication Logic.

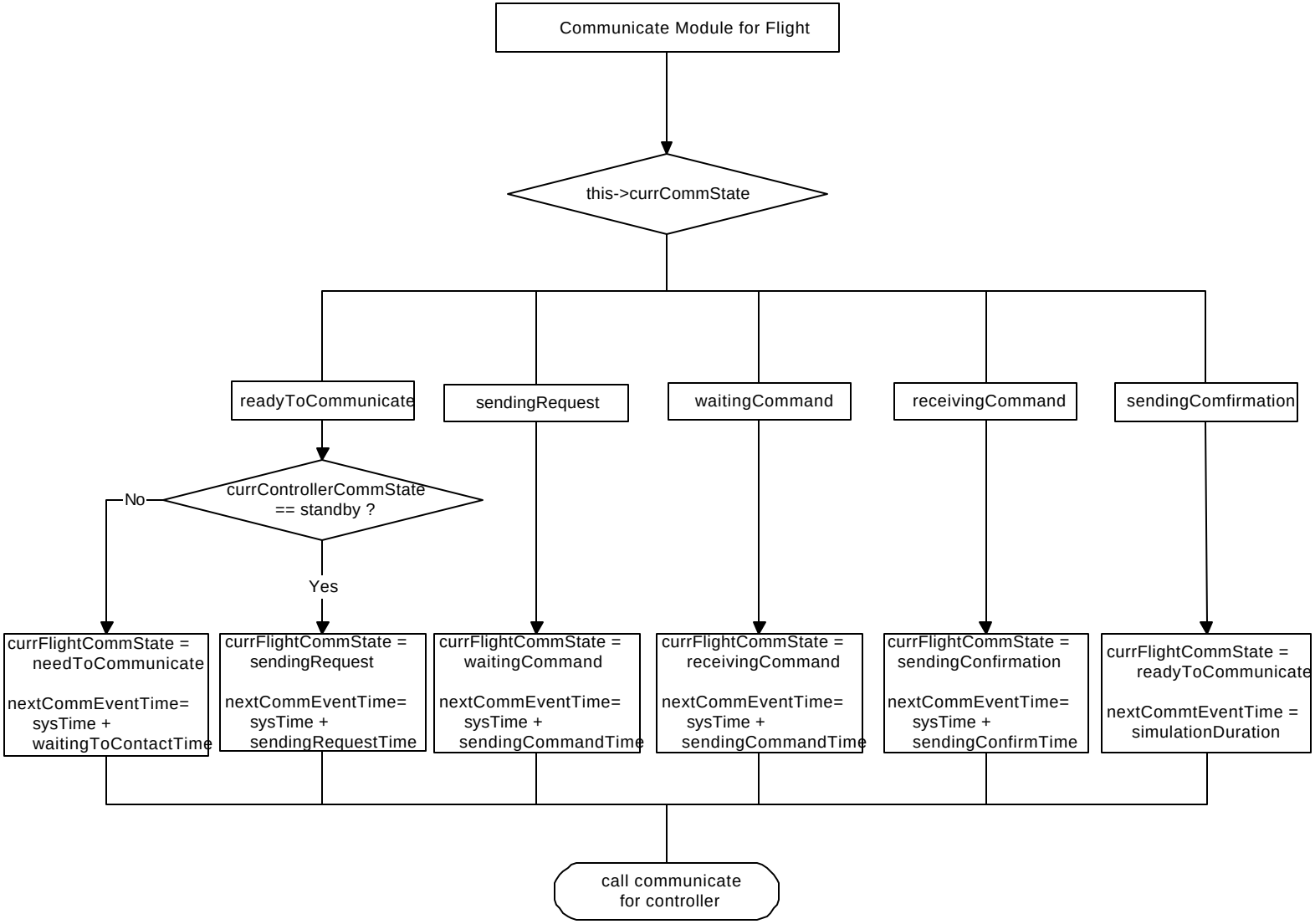


Figure 5.33 Flowchart for the Communication Module from the Flight's Point of View.

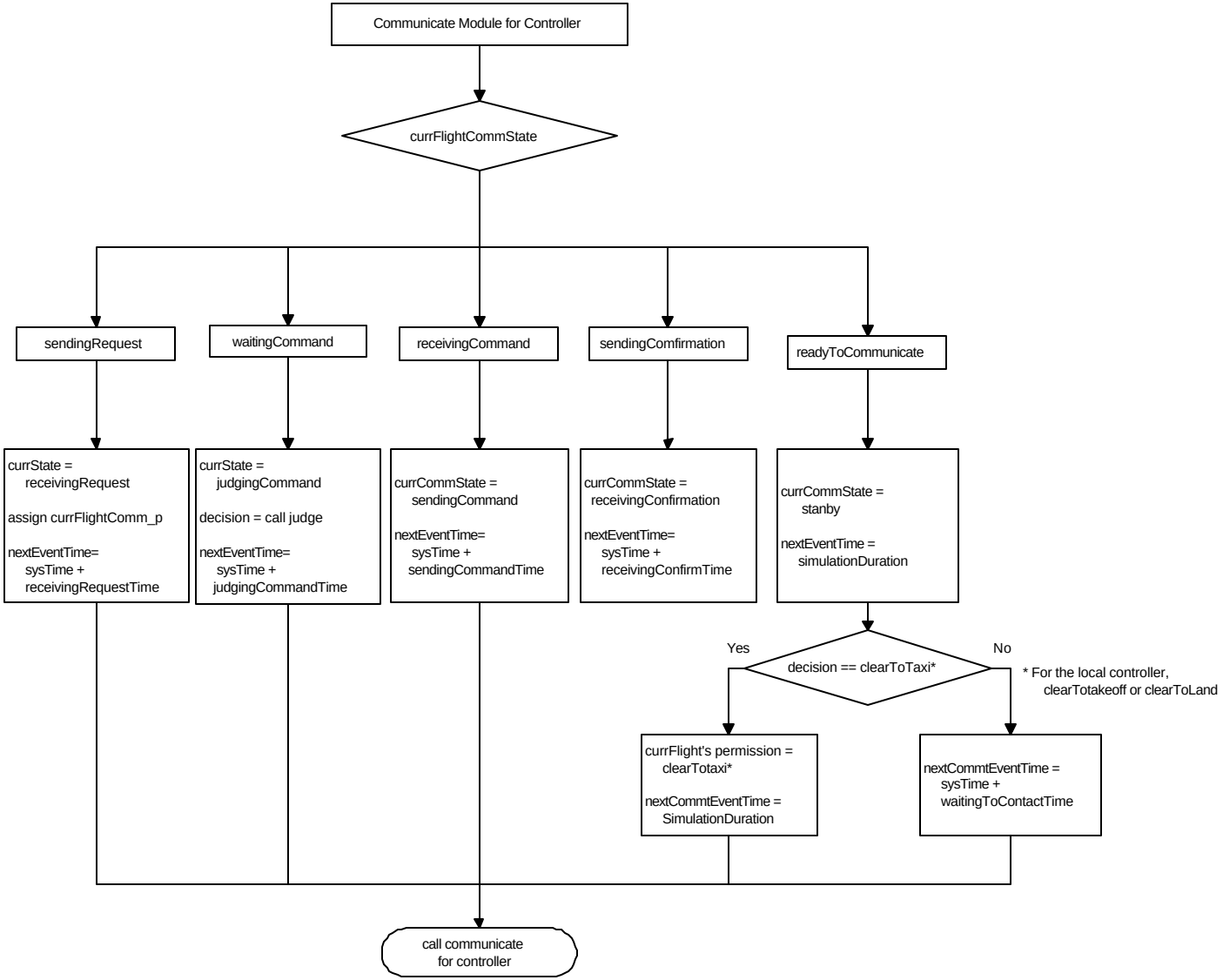


Figure 5.34 Flowchart for the Communication Logic from the Controller's Point of View.

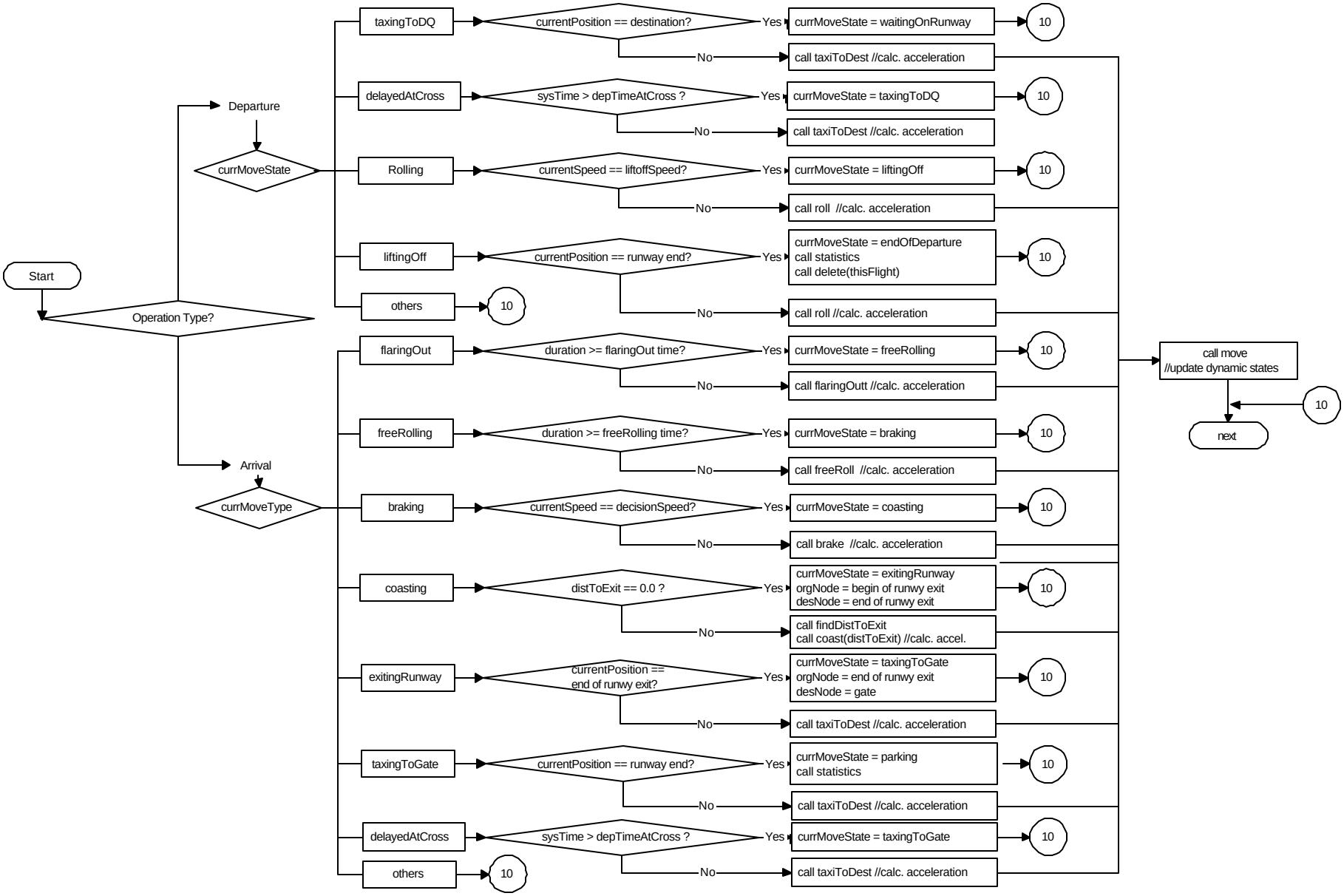


Figure 5.35 Flowchart for the Movement Logic.

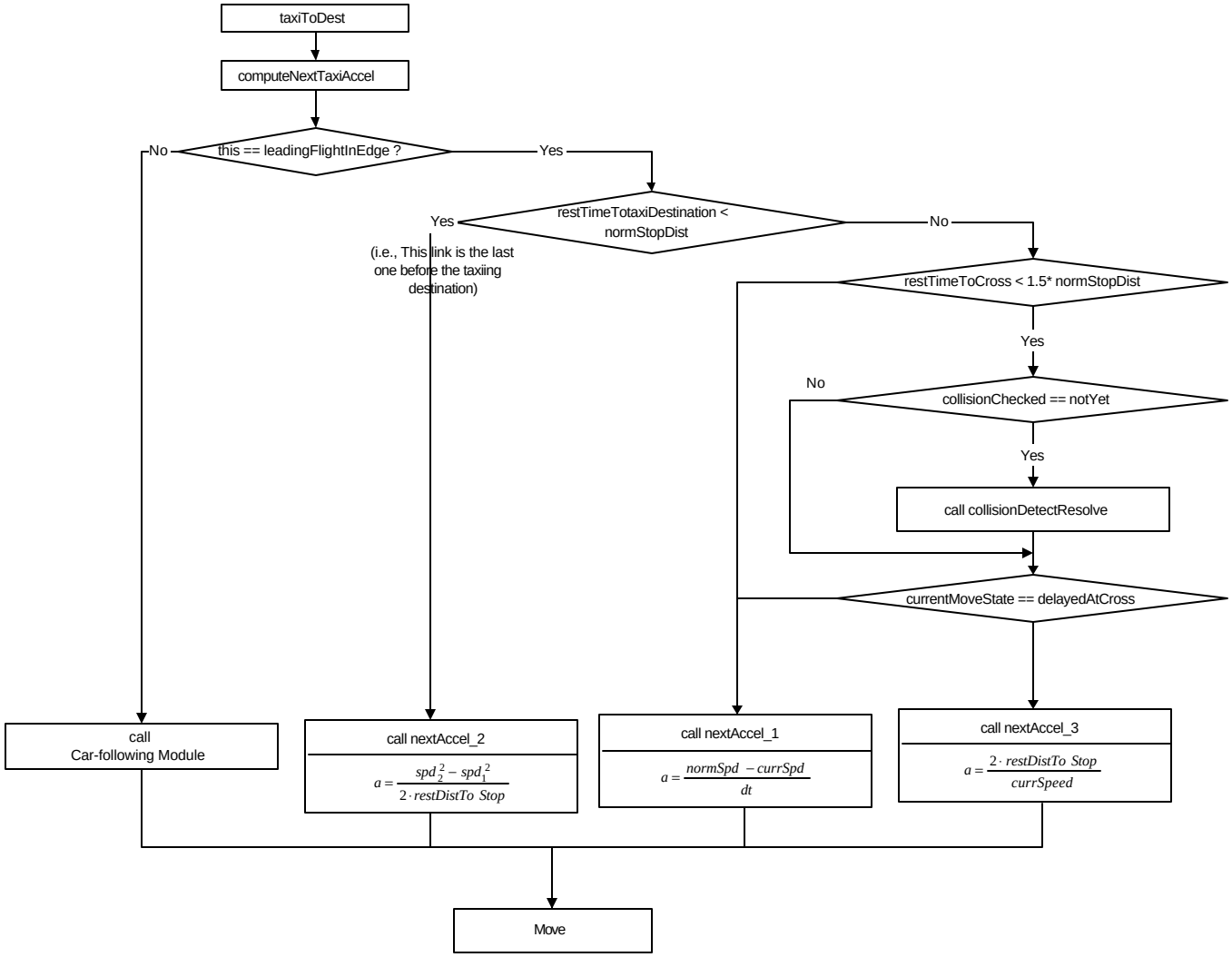


Figure 5.36 Flowchart for the Computation of the Next Acceleration for Taxiing.

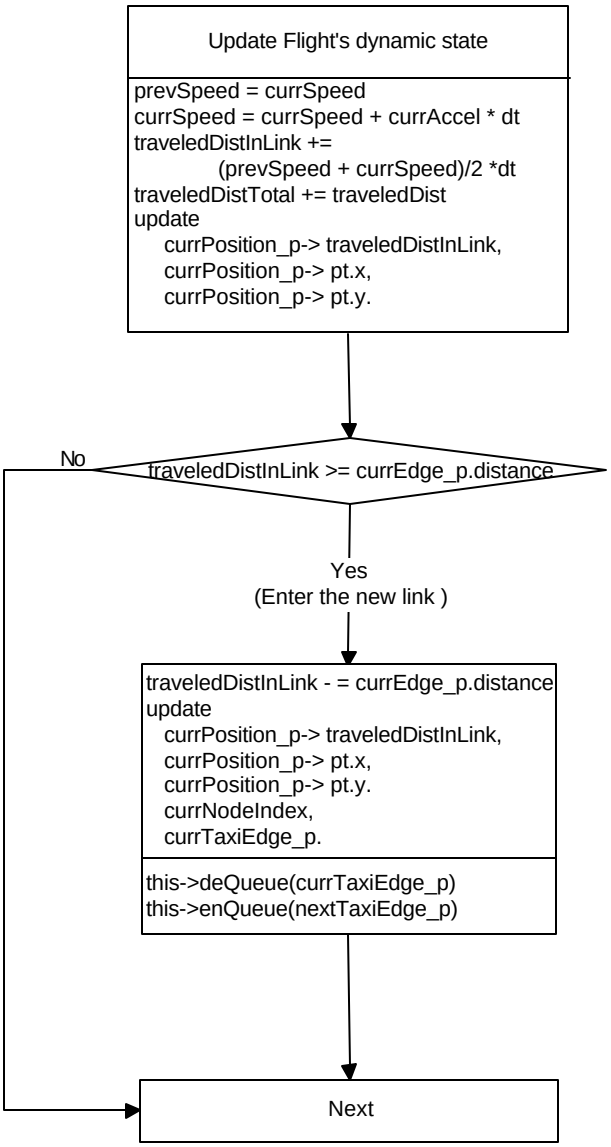


Figure 5.37 Flowchart for Updating Flight's Dynamic States.

5.3.2.4 Detailed Object Design Model

Once the algorithmic steps have been implemented using appropriate data structures, the model development cycle continues with a refinement of classes according to the following guidelines.

- Adjust class structures to increase inheritance: In order to increase inheritance, it is necessary to abstract out any behavior that is common to a group of classes. In our case, such behaviors as "finding shortest path" and "making taxiing path" are common to both the *ground controller* and the *local controller* classes. These methods are placed in the super class, *controller*. While the inheritance happens to be an "is-a" relationship, the aggregation is said to be a "has-a" relationship in the representation of classes. In the simulation model, many aggregation relationships are found. For example, a *node* class has a *point* class which contains information on the coordinates of a point, and an *edge* class which represents a taxiway link has two *node* classes corresponding to the *from node* and the *to node*. A *graph* class which contains the airport network information has an *edge list* consisting of a special number of *edges*.
- Design associations: Unlike inheritance or aggregation, the association between classes is often referred to as a "use-a" relationship. For example, the *controller* class uses (one or more) *queue* classes when a controller generates the shortest paths for all O-D pairs.

The attributes and member functions for all defined classes in the simulation model are summarized in Appendix C.

5.4 Object-oriented Programming (OOP)

The computer language selected for the simulation model is C++ which is one of the most popular object-oriented programming computer languages. In order to enhance the portability of the simulation model, all libraries used in the model are based on the ANSI C++ rather than using commercialized versions of the C++ libraries. In this research project, no graphic representation or GUI (Graphic User Interface) is considered. For the future development of a windowing GUI system, it is recommended to use the "wxWindow" library which provides a better multi-platform portability when compared to other commercial libraries such as the Microsoft Foundation Classes (MFC).

Once the simulation model is coded, the last step is the debugging and validation phase to check if the code is performing as expected. Several tips are recommended for debugging [May, 1990].

- Debug each member function separately.
- Use known deterministic data rather than stochastic data.
- Perform manual calculations to check.

5.5 Primary Validation of the Simulation Model

The validation step is the process whereby the simulation model is evaluated to determine whether it satisfactorily duplicates the real system behavior [Drew, 1966]. In order to match the simulation model output with real-life observations, a calibration process should be performed using collected input data and measures of effectiveness (MOE). In this research, the calibration process considers accepted models of behavior and compares them with the output from the simulation model. VTASM is a proof-of-concept research model and further validation is needed for commercial use.

Table 5.8 shows a hypothetical flight schedule consisting of 20 flights scheduled in about 5 minutes. Here, the relatively tight flight schedule is intentional to produce conflicts on the taxiway system. It is assumed that all flights use one runway 01, and all aircraft represent transport type operations. The aircraft mix is set to be 70/30 (large/small).

Important simulation outputs include information on the flight states at every simulation interval, such as movement state, current communication state, speed, acceleration, current position, traveled distances etc. Figure 5.38 shows a sample output of the simulation results. The time-space diagram shown in Figure 5.39 depicts the behavior of four departing aircraft starting from one common gate to the same runway.

A close examination of this figure confirms that headways between the aircraft are kept during the taxiing phase, and the minimum separation rules for the takeoff aircraft are maintained. A more detailed takeoff profile is illustrated in Figure 5.40. Figure 5.41 also ratifies the minimal separations between successive landing aircraft. It is shown in Figure 5.42 that landing distances and runway occupancy times presented in Table 5.3 are reproduced with good accuracy.

For the aircraft-following model defined in Equation (5.2), H_j and v_f are set to 83 ft/aircraft, 41 ft/s respectively. Figure 5.43 shows that headway between two leading and following aircraft correlates well with the speed and vehicle-following control laws stated in the Equation (5.2). In this Figure, some discrete speeding phenomena is observed following the aircraft transition through various taxiway links. This behavior is explained when the following aircraft returns to a normal taxiing speed, once it becomes the leading aircraft on a link.

Table 5.8 Hypothetical Flight Schedule.

Flight #	Aircraft Type	Operation Type	Schedule Time			Gate	Runway
			Hour	Minute	Second		
AA001	B727-100	D	7	0	0	1	36
AA002	B727-100	D	7	0	4	1	36
AA003	SF340	D	7	0	20	2	36
AA004	B727-100	D	7	0	40	2	36
AA005	B727-100	D	7	0	50	3	36
AA006	B727-100	D	7	0	30	3	36
AA007	SF340	D	7	0	10	4	36
AA008	B727-100	D	7	0	45	4	36
AA009	B727-100	D	7	0	25	4	36
AA010	SF340	D	7	0	30	4	36
AA101	B727-100	A	7	1	0	11	36
AA102	B727-100	A	7	3	1	11	36
AA103	B727-100	A	7	3	2	12	36
AA104	SF340	A	7	3	3	12	36
AA105	B727-100	A	7	1	12	10	36
AA106	B727-100	A	7	1	45	10	36
AA107	SF340	A	7	1	20	10	36
AA108	SF340	A	7	5	3	9	36
AA109	B727-100	A	7	3	0	9	36
AA110	B727-100	A	7	2	1	9	36

Time	FLT #	(x, y)	currCommState	currMoveState	speed	accel	currLink	linkLength	distInLink	totDist
60.5	AA378	(3.55881,6.15414)	readyToCommunicate	taxiingToDepQue	3.20622	15.6622	4 -> 1016	157.659	0.324311	0.324311
	AA789	(3.55850,6.15410)	readyToCommunic ate	parking	0.00000	0.0000	0 -> 0	0	0.000000	0.000000
	AA790	(3.55850,6.15410)	readyToCommunicate	parking	0.00000	0.0000	0 -> 0	0	0.000000	0.000000
	AA791	(3.55850,6.15410)	readyToCommunicate	parking	0.00000	0.0000	0 -> 0	0	0.000000	0.000000
60.6	AA378	(3.55918,6.15418)	readyToCommunicate	taxiingToDepQue	4.70198	14.9576	4 -> 1016	157.659	0.719721	0.719721
	AA789	(3.55850,6.15410)	readyToCommunicate	parking	0.00000	0.0000	0 -> 0	0	0.000000	0.000000
	AA790	(3.55850,6.15410)	readyToCommunicate	parking	0.00000	0.0000	0 -> 0	0	0.000000	0.000000
	AA791	(3.55850,6.15410)	readyToCommunicate	parking	0.00000	0.0000	0 -> 0	0	0.000000	0.000000

Figure 5.38 Preliminary Results (1): Sample Output.

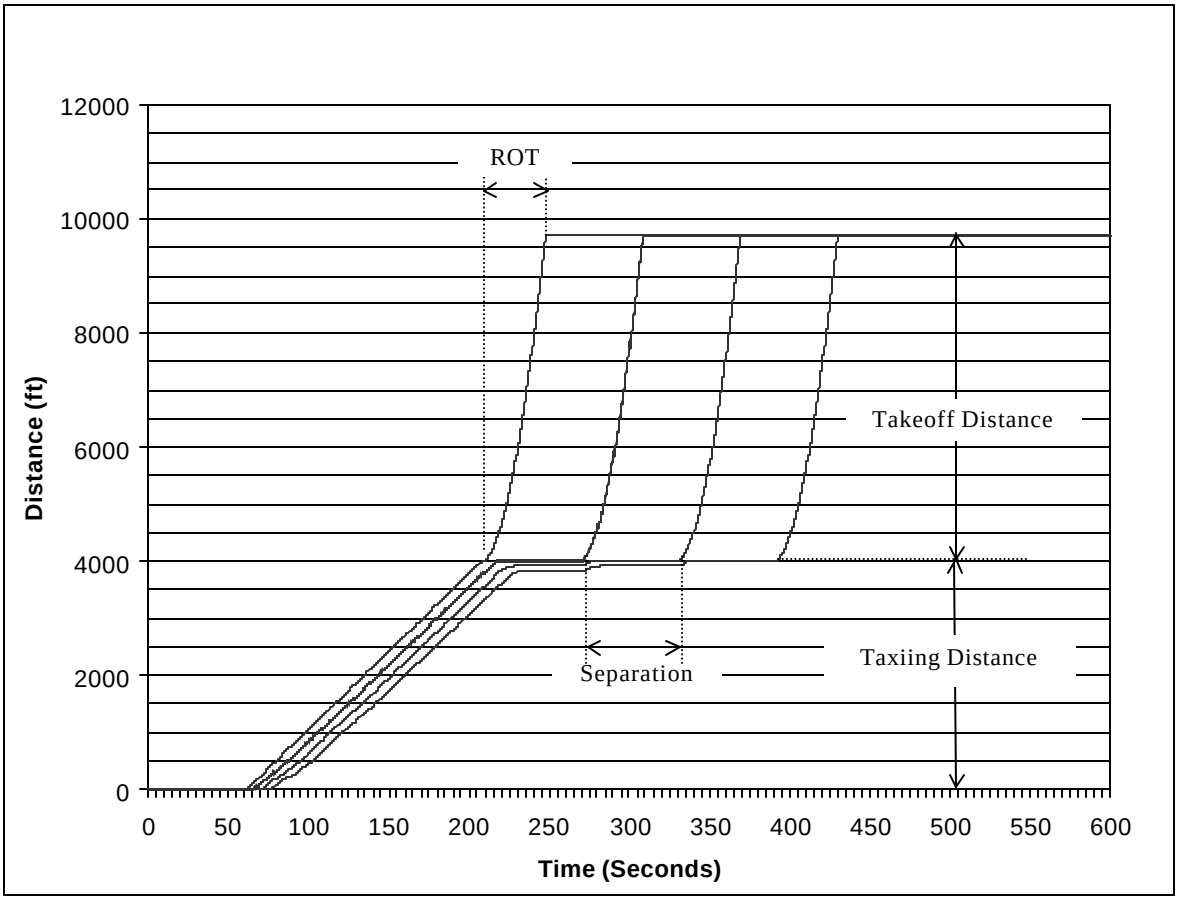
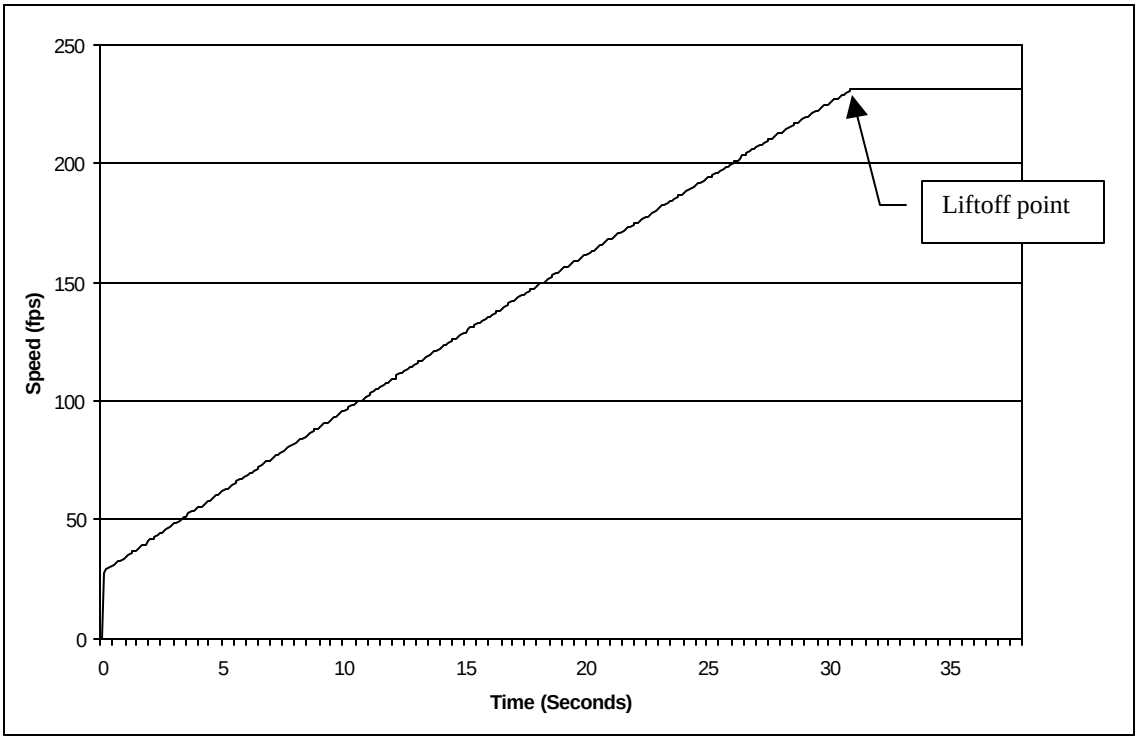


Figure 5.39 Validation of the Simulation Model (Time-Space Diagram for Departures, B727-100).



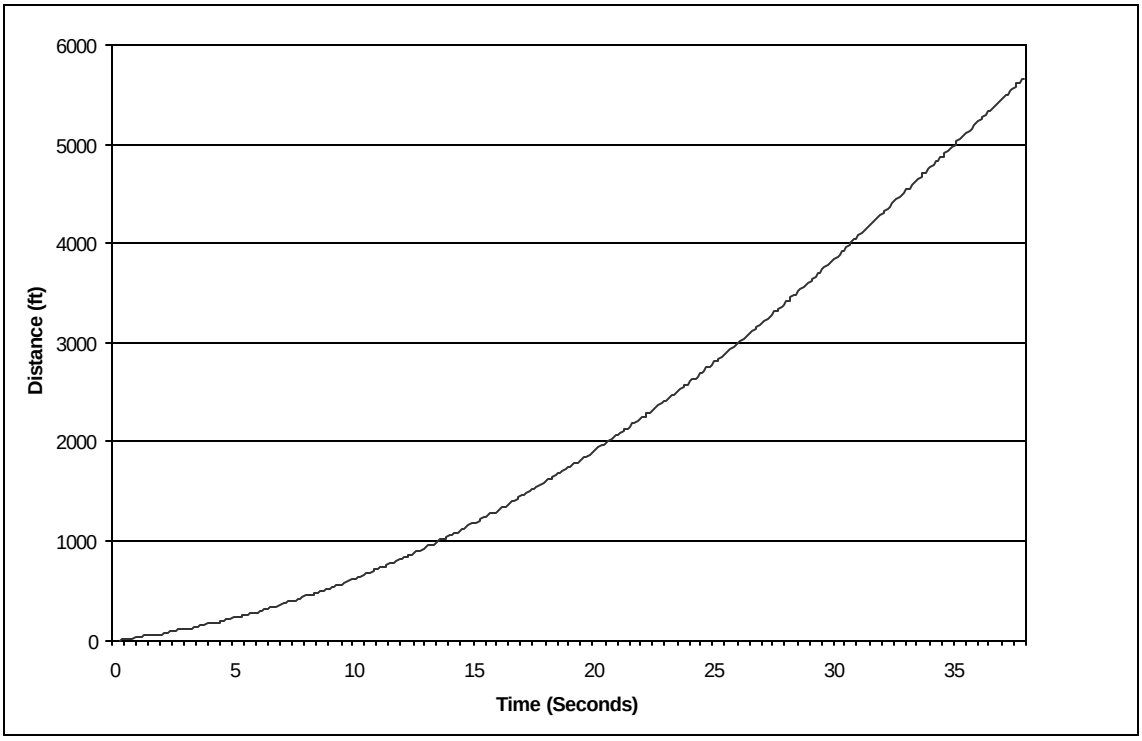


Figure 5.40 Validation of the Simulation Model (Takeoff Profile, B727-100).

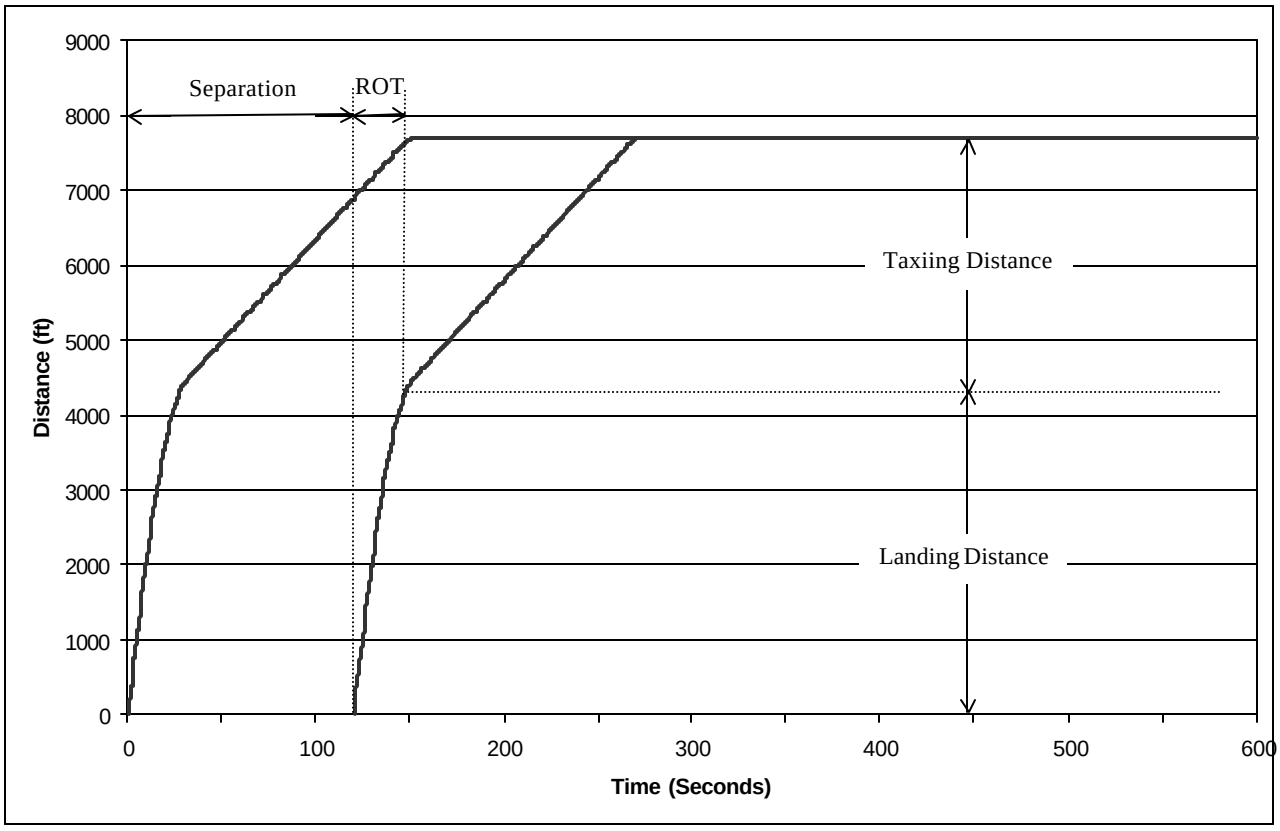


Figure 5.41 Validation of the Simulation Model (Time-Space diagram For Arrivals, B727-100).

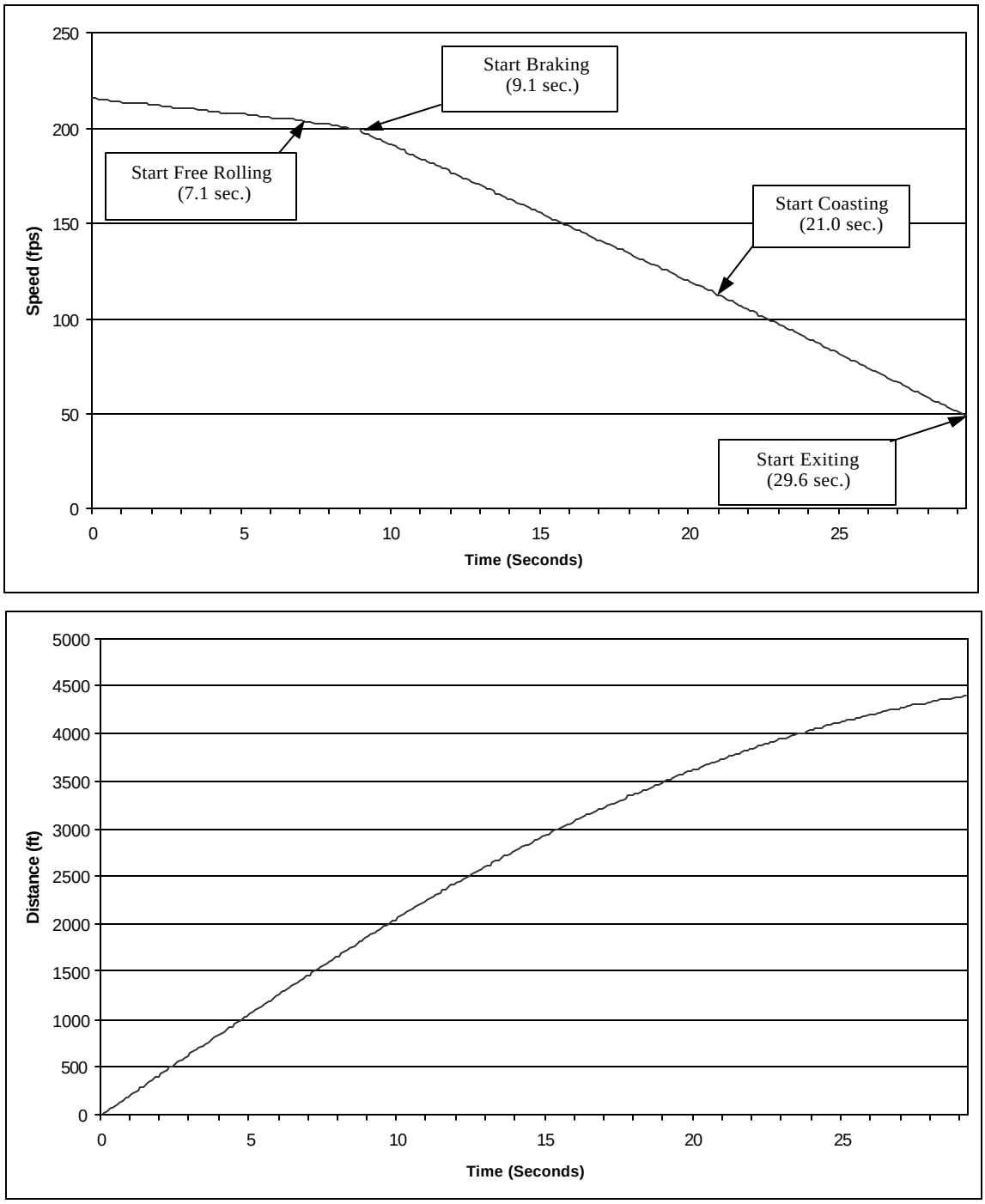


Figure 5.42 Validation of the Simulation Model (Touchdown Profile, B727-100).

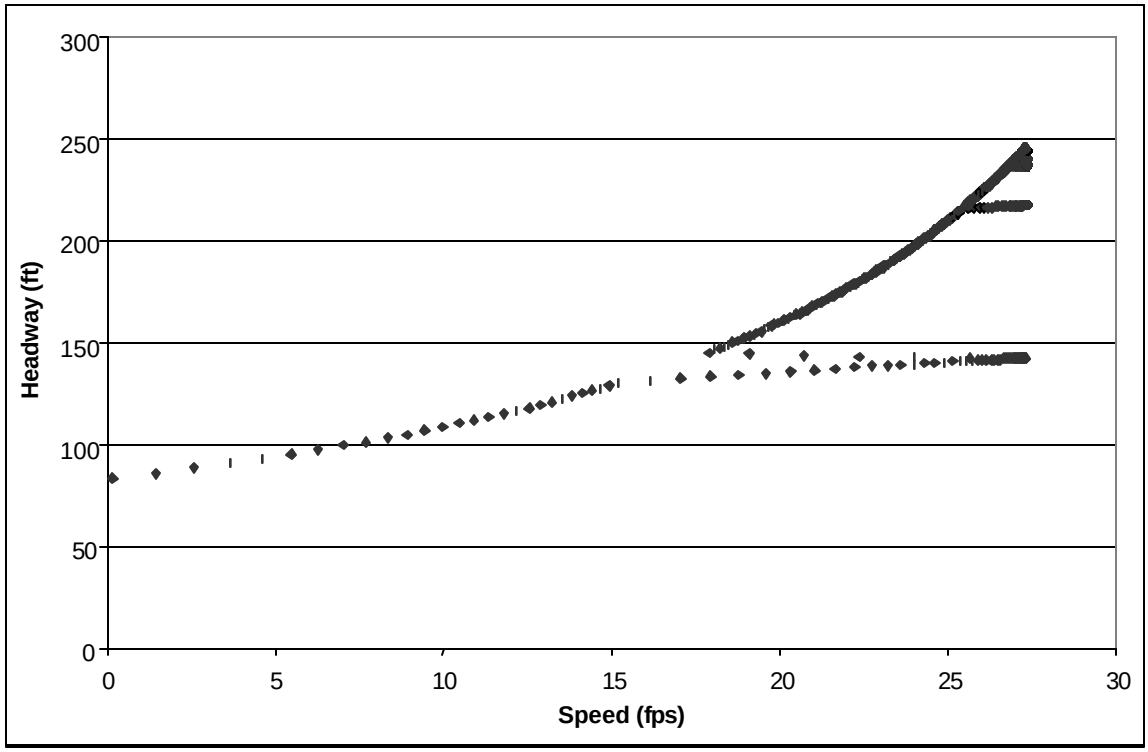


Figure 5.43 Validation of Simulation Model (Headway-Speed Profile, B727-100).

Chapter 6. Case Study

One of the main benefits of a microscopic simulation model is the wealth of information derived from each simulation run. In this section, we first define several types of delays encountered in airport networks. A case study is then presented to illustrated the use of the model developed.

6.1 Definition of Delays

The Consolidated Operations and Delay Analysis System (CODAS) prepared by the FAA defines three types of delays involved in airport operations [FAA, 1997].

- *Gate delay*: the difference between the actual gate departure time reported in ASQP (Airline Service Quality Performance) system and the scheduled gate departure time reported in ETMS (Enhanced Traffic Management System).
- *Taxi-out delay*: The difference between the actual taxi-out time (= wheels-off time – gate-out time) and the unimpeded taxi-out time at the airport. The unimpeded taxi-out time is the estimated average taxi-out time for an aircraft under optimal operating conditions when neither congestion, weather, or other factors delay the operation during its movement from the gate to takeoff.
- *Taxi-in delay*: The difference between the actual taxi-in time (= gate arrival time – wheels-on time) and the unimpeded taxi-in time under an unimpeded condition.

The taxi-out delays defined in CODAS involve not only the delays due to the taxiway congestion itself, but also delays due to the excess demand on runway operations. On the other hand, taxi-in delay does not contain runway delays, but only consider taxiway delays. This shows a discrepancy between taxi-in and taxi-out delays. Motivated by this point, let us define a single delay called *total delay* applicable to both arriving and departing operations. The *total delay* is defined as follows (See Figures 6.1 and 6.2):

- *Total delay*: the difference between two completion times of the nominal (or unimpeded) operation time and actual operation time. Here, the completion time of nominal operation is

the time duration which is needed for a flight to complete its operation impeded by no other operations.

Even though it is not easy to divide the total delay into various sub-types, for simulation purposes, the taxiing delay and runway delays are defined as follows:

- *Runway delay*: The difference between the time when an aircraft is scheduled to start its runway operation and the time when the actual operation takes place.
- *Taxiing delay*: During taxiing duration, the taxiing delay accrues whenever any flight's taxiing speed is less than a nominal taxiing speed. The taxiing delay is estimated by the following equation.

$$\text{Taxiing delay} = \int_0^{\text{simulationtime}} \left(\frac{\text{nominal speed} - \text{current speed}}{\text{nominal speed}} \right) dt .$$

Taxiing duration (or *taxiing time*) is defined as the time required for a departing aircraft to taxi from the gate to the runway departure queue, For an arriving aircraft is the time to taxi from the runway exit to the gate.
Figures 6.1 and 6.2 illustrate the various types of delays for both arriving and departing flights.

6.2 Sample Airport

In this research project, the Ronald Reagan National Airport (DCA) is selected for further study. Along with Dulles Airport and Baltimore-Washington Airports, DCA serves the Washington D.C. metropolitan area. In FY 1997, total enplanements and operations at DCA were 7,231,903 and 316,404, respectively, placing it as the 26th busiest airport in the U.S. [FAA, 1998].
The existing DCA Airport has 45 gates and three crossing runways designated 3/21 (4,506 ft), 15/33 (5,189 ft), and 18/36 (6,869 ft). (As February 2000, Runway 18/36 has been changed to 01/19 due to magnetic declination.) Because of the relatively short length of its runways, DCA has short to medium size transport aircraft operations. Figure 6.3 shows the present configuration of DCA. In this preliminary analysis, the gates are aggregated into 12 groups and it is assumed that several flights can occupy one aggregated gate at the same time. (This modeling practice is common in other simulation models.)

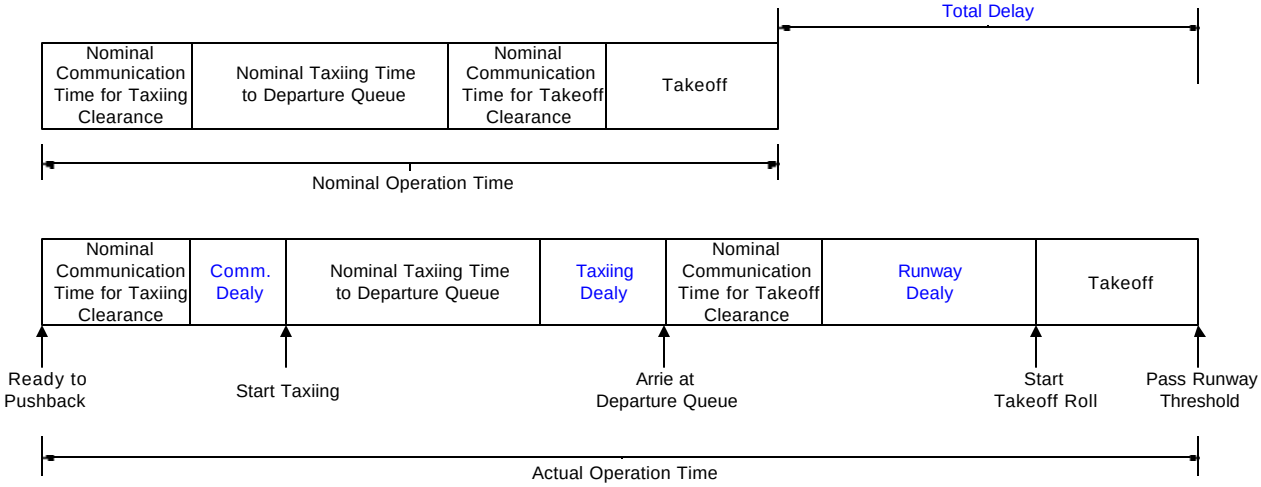


Figure 6.1 Delays Associated with a Departing Flight.

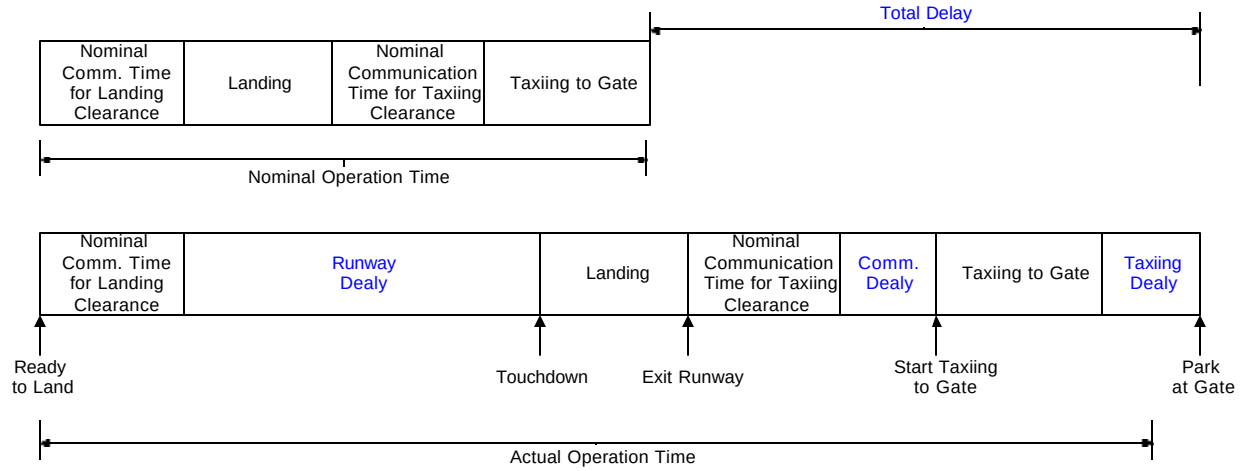


Figure 6.2 Delays Associated with an Arriving Flight.

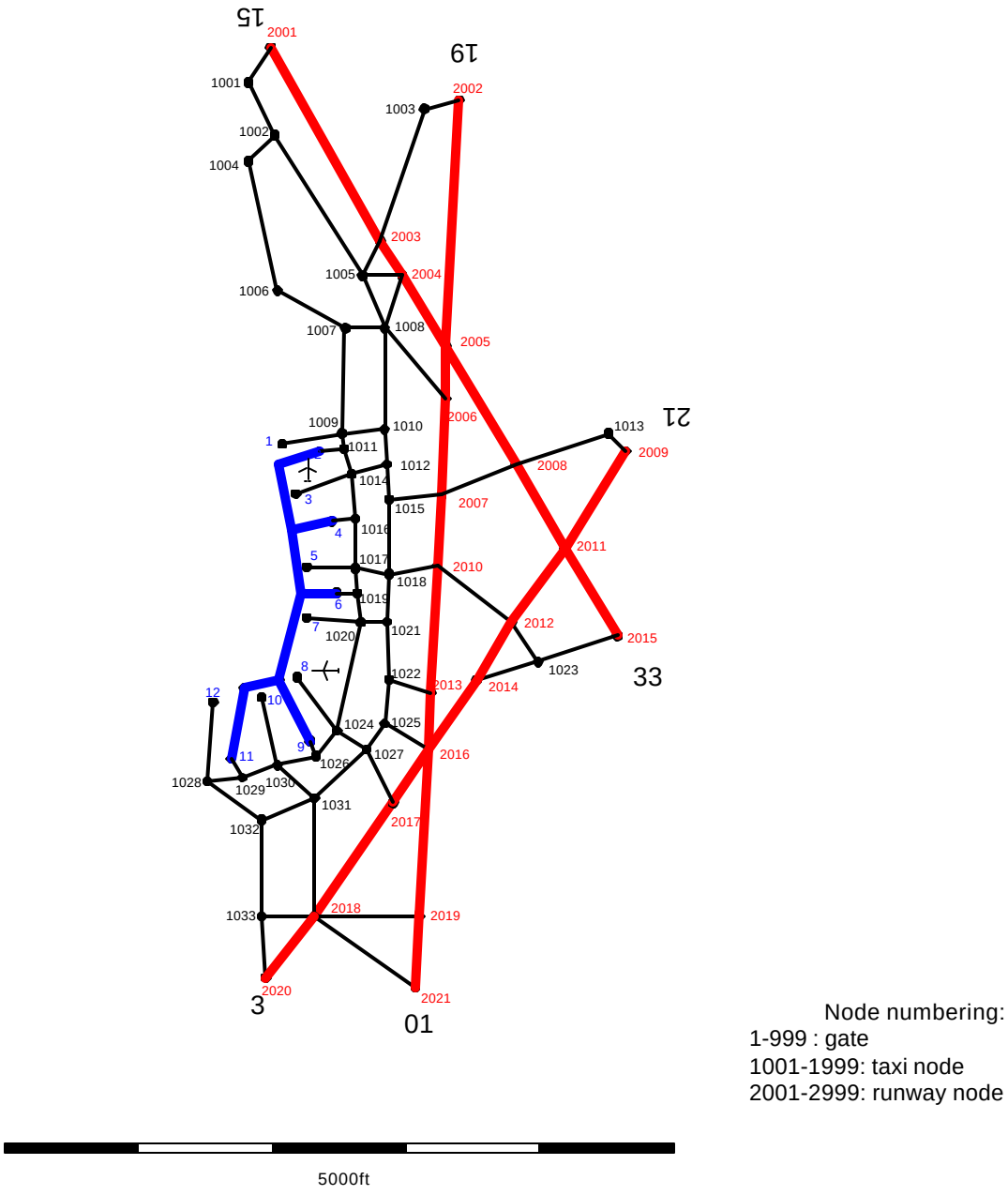


Figure 6.3 Configuration of the Washington National Airport (DCA).

6.3 Scenarios

To evaluate the efficiency of the airport control system, a total 12 scenarios are generated through variations in three important decision variables: 1) Network assignment strategy, 2) sequencing strategy, and 3) pilot-controller communication method. Two network assignment strategies are considered: static and time-dependent assignment methods. The static network assignment uses the Euclidean distances for the shortest path algorithm. The sequencing strategy involves pure first-come-first-serve (FCFS), FCFS with landing priority, and 2-exchange sequences. The impact of two types of pilot-controller communication methods are also integrated using standard voice channel and data link methods.

A base line scenario attempts to reproduce the current ATC system using principle of FCFS with landing priority. In this scenario, both the local and the ground controllers use standard voice channel communications to provide traffic control advisories. In the scenario, a static assignment method is used for ground controller to find the taxiing route.

The most advanced scenario studied represents an advanced ATC system where the local controllers follow optimal aircraft sequences using results from the ASP (Aircraft Sequencing Problem) model, and the ground controllers guide the taxiing flights based on dynamic shortest paths which result from NAP (Network Assignment Problem). All 12 scenarios are summarized in Table 6.1.

Table 6.1 Scenarios for Case Study.

Network Assignment Strategy	Sequence Method	Communication Method.
Static Network Assignment	Pure FCFS	Voice Channel
		Data Link
	FCFS (w/ Landing Priority)	Voice Channel (**Base line scenario**)
		Data Link
	2-exchange	Voice Channel
		Data Link
Time-dependent Network Assignment	Pure FCFS	Voice Channel
		Data Link
	FCFS (w/ Landing Priority)	Voice Channel
		Data Link
	2-exchange	Voice Channel
		Data Link

6.4 Computational Process

The computations for the case study are derived using the following steps: Data generation, sequencing, simulation, and the analysis of results. Figure 6.4 illustrates a flowchart with all computations along with data and output for each step.

6.4.1 Data Generation

Using random number generation techniques, the ready-times (or nominal times) for runway operations are generated. A total of nine levels of hourly demand (ranging from 10 to 50 flights per hour) are generated. To consider randomness of generated data, ten data sets are generated for each level of hourly demand. In all test scenarios, the interval for any two consecutive runway operations (either takeoff or landing) follows a negative exponential distribution. The aircraft mix which is another important factor in the delay analysis, is set to 0% heavy, 70% large and 30% small. For computational simplicity, it is assumed that all flights use a single runway 01. (This issue will be discussed later in further study.)

Once nominal times for runway operations are available, the activation times for departing flights are obtained by subtracting the sum of nominal taxiing, communication and some buffer times from nominal times. The link and node data which are pertinent to taxiway topology are used to obtain nominal taxiing times. Unlike departing flights, the activation times for arriving flights are obtained by subtracting only communication times from nominal times.

6.4.2 Sequencing

In this step, three types of sequences for runway operations are computed using nominal times: pure FCFS sequence, FCFS sequence with landing priority and 2-exchange sequence. Table 6.2 shows a sample output of the sequencing process. In the sequencing process, a maximum of 600 seconds of delay is assumed tolerable as practical limit. This limit can be altered by a planner and applied for each flight time window. In fact, the schedule from a pure FCFS sequence is the same one as that of nominal times.

Table 6.2 also shows two types of time savings attained if the swapped sequence is applied. These are: cumulative time savings obtained from all individual flights, and time savings in completion time of all flights. For example, the table shows that if the runway operations are performed in

accordance with a 2-exchange sequence instead of the FCFS sequence with landing priority, a total of 669 seconds can be saved over all flights. In this case, the completion time for all operations are reduced from 1243 to 1149 seconds.

Table 6.2 A Sample Output from the Sequencing Model.

#	acft_id	Type	Nominal ReadyTime (1)	DueTime	FCFS w/ landing priority StartTime (2)	Sequence	Delay =(2)-(1)	2-exchange StartTime (3)	Sequence	Delay =(3)-(1)	Saving =(3)-(2)
1	DEP_1	L	[186	786]	268	4	82	268	4	82	0
2	DEP_2	S	[218	818]	375	6	157	375	6	157	0
3	DEP_3	S	[302	902]	482	8	180	594	10	292	-112
4	DEP_4	L	[322	918]	589	10	267	496	8	174	93
5	DEP_5	L	[378	978]	800	13	422	747	13	369	53
6	DEP_6	S	[414	1014]	907	15	493	687	12	273	220
7	DEP_7	L	[444	1044]	1028	17	584	840	15	396	188
8	DEP_8	L	[536	1123]	1088	18	552	947	17	411	141
9	DEP_9	L	[583	1154]	1148	19	565	1054	19	471	94
10	DEP_10	L	[614	1214]	1208	20	594	1114	20	500	94
11	ARR_1	L	[17	582]	17	1	0	17	1	0	0
12	ARR_2	S	[83	683]	148	2	65	148	2	65	0
13	ARR_3	L	[89	648]	228	3	139	228	3	139	0
14	ARR_4	L	[155	755]	335	5	180	335	5	180	0
15	ARR_5	L	[269	800]	442	7	173	647	11	378	-205
16	ARR_6	L	[307	851]	549	9	242	800	14	493	-251
17	ARR_7	S	[352	952]	680	11	328	466	7	114	214
18	ARR_8	L	[358	924]	760	12	402	907	16	549	-147
19	ARR_9	L	[431	1031]	867	14	436	1014	18	583	-147
20	ARR_10	S	[468	1068]	998	16	530	564	9	96	434
			Total Delay:				6391			5722	669

Completion time: FCFS w/land -> 1243 second, SWAP -> 1149 second

It is important to notice that some flights might not be operated as they are scheduled. This is because the resulting schedules are an analytical solution which do not reflect any congestion in communication or taxiing. As pointed out at the beginning of this research, it is very difficult to develop a single analytical model which considers all of types of delays. This is the main justification for using a simulation model.

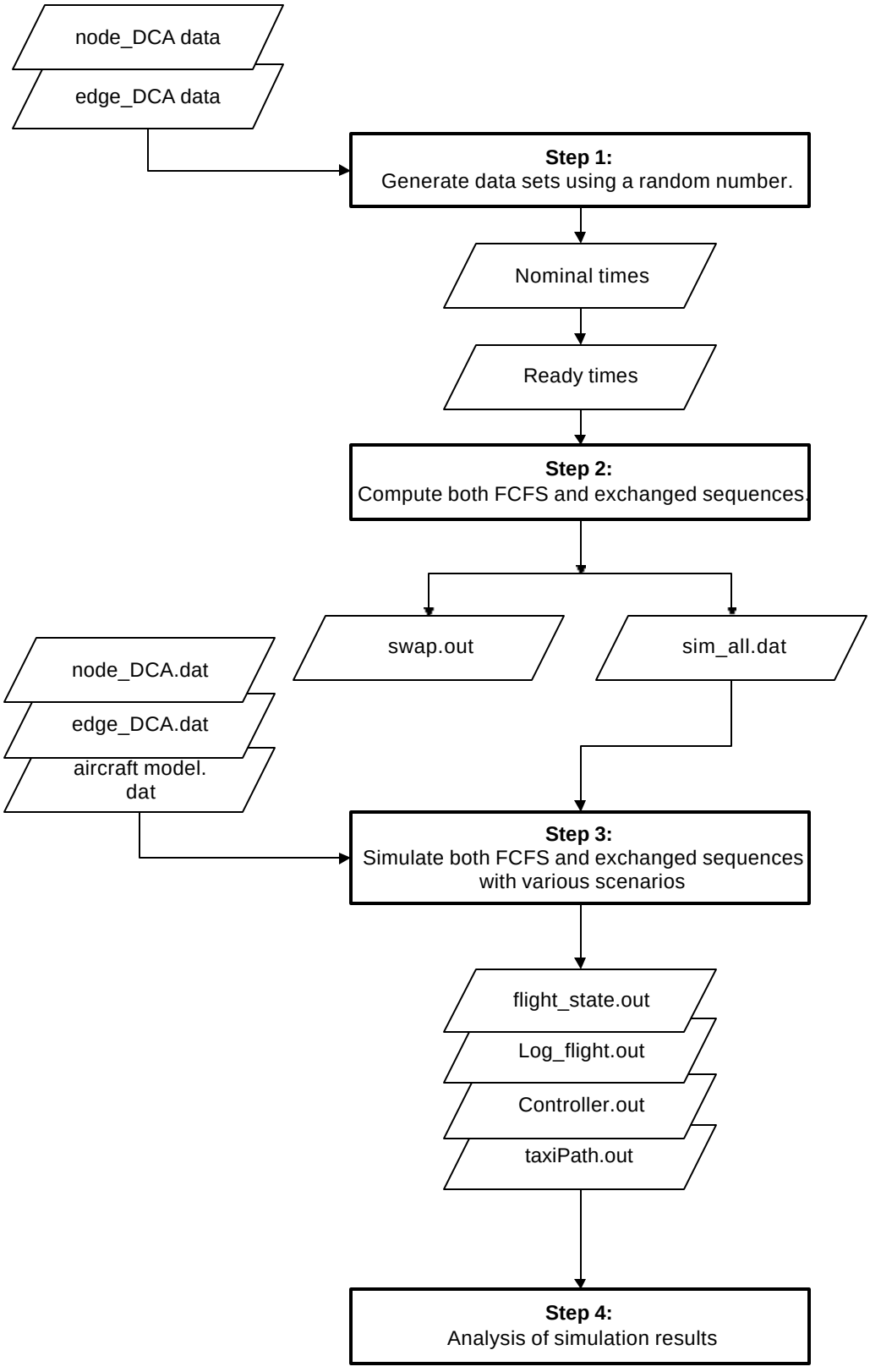


Figure 6.4 Flowchart of the Computational Process Employed in the Case Study.

6.4.3 Simulation

6.4.3.1 Assumptions

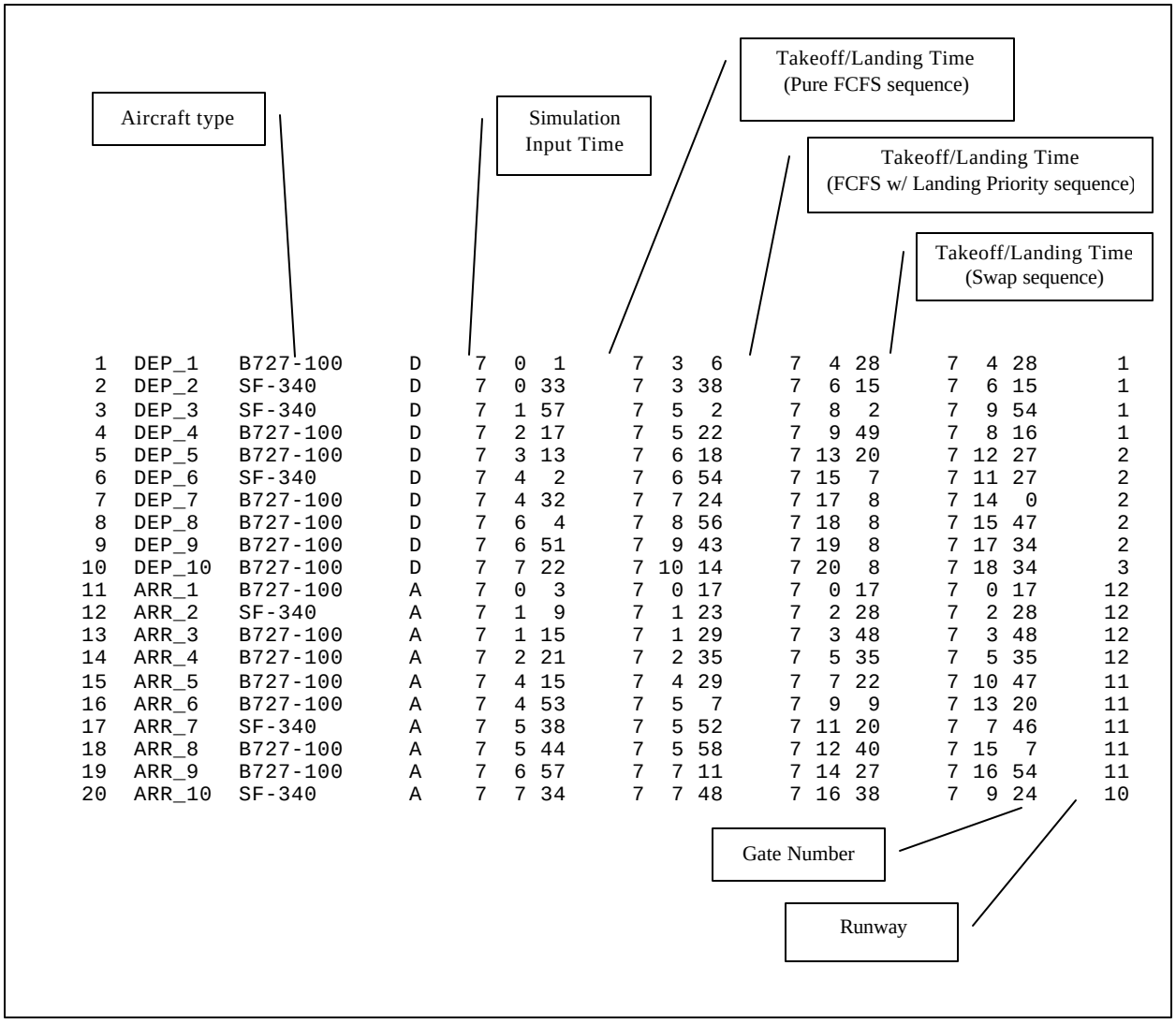
The VTASM model has numerous constants representing human or aircraft behaviors. For example, the free flow speed, normal speed and jam headways employed in the aircraft-following model are set to 45 km/hr, 30 km/hr and 27 m, respectively. It should be pointed out that, since some of these constant values have been drived using common sense, more accurate values could be collected from future field studies. Some of important constants used in the simulation model are summarized in Table 6.3.

Table 6.3 Constant Values used in the VTASM Simulation Model.

Related model	Name	Values
Aircraft following Model	Maximum taxiing speed	45 (km/hr)
	Normal taxiing speed	30 (km/hr)
	Jam headway	27 (m)
Flaring out	Gamma	2.75 (degree)
	Runway threshold crossing height	15 (m)
	Flare load factor	1.1
	Weight factor for landing	0.5
Free rolling	Time for free rolling	2.0 (s)
	Acceleration for free rolling	-0.7 (m/s^2)
Braking	Weight for decision speed	18450 (kg) = 41000 (lb)
	Decision speed for heavy aircraft	35 (m/s)
	Decision speed for large or small aircraft	27 (m/s)
Runway exit	Exit speed for normal runway exit	15 (m/s)
Communication (Voice channel)	Sending request time (same to receiving request time for controller)	4.0 (s)
	Waiting command time (same to receiving request time)	5.0 (s)
	Receiving command time (same to judging time for controller)	3.0 (s)
	Sending confirmation time (same to receiving request time for controller)	3.0 (s)
	Waiting time for next contact	10.0 (s)
Communication (Data Link)	Sending request time	0.0 (s)
	Waiting command time	3.0 (s)
	Receiving command time	0.0 (s)
	Sending confirmation time	0.0 (s)
Link travel time function	current link travel time + 5 seconds for every one conflicting aircraft at the intersecting point	

6.4.3.2 Input Data

Input data for simulation model include link and node information representing the taxiway and runway configurations, the aircraft model data, and the flight schedule data. Aircraft model data involves individual aircraft information such as size, weight and performance data. The flight schedule data reflects the results of the previous sequencing step. Using the given schedule data, recommended pushback times for departing flights are internally computed. (For a detailed description about the recommended pushback times, see Section 2.3.1.) A sample of schedule is shown in Figure 6.5.



1	DEP_1	B727-100	D	7	0	1	7	3	6	7	4	28	7	4	28	1
2	DEP_2	SF-340	D	7	0	33	7	3	38	7	6	15	7	6	15	1
3	DEP_3	SF-340	D	7	1	57	7	5	2	7	8	2	7	9	54	1
4	DEP_4	B727-100	D	7	2	17	7	5	22	7	9	49	7	8	16	1
5	DEP_5	B727-100	D	7	3	13	7	6	18	7	13	20	7	12	27	2
6	DEP_6	SF-340	D	7	4	2	7	6	54	7	15	7	7	11	27	2
7	DEP_7	B727-100	D	7	4	32	7	7	24	7	17	8	7	14	0	2
8	DEP_8	B727-100	D	7	6	4	7	8	56	7	18	8	7	15	47	2
9	DEP_9	B727-100	D	7	6	51	7	9	43	7	19	8	7	17	34	2
10	DEP_10	B727-100	D	7	7	22	7	10	14	7	20	8	7	18	34	3
11	ARR_1	B727-100	A	7	0	3	7	0	17	7	0	17	7	0	17	12
12	ARR_2	SF-340	A	7	1	9	7	1	23	7	2	28	7	2	28	12
13	ARR_3	B727-100	A	7	1	15	7	1	29	7	3	48	7	3	48	12
14	ARR_4	B727-100	A	7	2	21	7	2	35	7	5	35	7	5	35	12
15	ARR_5	B727-100	A	7	4	15	7	4	29	7	7	22	7	10	47	11
16	ARR_6	B727-100	A	7	4	53	7	5	7	7	9	9	7	13	20	11
17	ARR_7	SF-340	A	7	5	38	7	5	52	7	11	20	7	7	46	11
18	ARR_8	B727-100	A	7	5	44	7	5	58	7	12	40	7	15	7	11
19	ARR_9	B727-100	A	7	6	57	7	7	11	7	14	27	7	16	54	11
20	ARR_10	SF-340	A	7	7	34	7	7	48	7	16	38	7	9	24	10

Figure 6.5 Sample of Schedule Data.

6.4.3.3 Implementation of the Simulation Model

Each simulation is executed for two-hour duration (i.e., 7200 seconds). This is done to secure enough time for all flights to finish their operations during the simulation duration. The system clock is set to advance by one second. This time interval, *dt*, is one of important settings particularly in association with aircraft-following model. There is a tradeoff in selecting the size of *dt*. Smaller *dt* provides more detailed results in aircraft behavior but requires more time and storage space in the implementation simulation.

For the time-dependent NAP, a total of 120 time slices are prepared so that the time-dependent shortest path information can reflect any change in link travel time with a resolution of 60 seconds. The simulation is repeated for all 12 scenarios.

6.4.3.4 Output Files

VTASM provides several output files for further scrutiny of the data.

1) 1) *Log.out* reports important events for both flights and controllers including a summary report with statistics.
2) 2) *FlightState.out* includes the flight state at each time slice.
3) *Controller.out* contains the controller state at each time slice.
4) *ForwardStar.out/backwardStar.out* shows forward-star/backward-star of network configuration (used for verification).
5) *TaxiPath.out* contains static and time-dependent taxi paths for all flights (used for verification).

The summary section in the *log.out* file reports diverse statistics related to the delay analysis for each flight such as taxiing duration, runway occupancy time, runway delay, etc. The flight state in file *flight_state.out* includes position, communication state, movement state, permission, speed, acceleration and traveled distance at time *t*. Figures 6.6 and 6.7 show samples of the *log,out* and the *flight_state.out* files. A sample of the *path.out* file is illustrated in Figure 6.8. In this figure, it is shown that some flights have different paths in the static and in the time-dependent NAP.

```
Data files were opened.

FS and BS were made.
    FLight DEP_1 is pushed back to GC's PENDING list
    FLight DEP_2 is pushed back to GC's PENDING list
...
    FLight ARR_10 is pushed back to LC's PENDING list
    Flight data were read.
Shortest Path was made.

< Simulation starts >
    system clock = 0 second
                 At 1 sec, DEP_1 is removed from GC's PENDING list
                 At 1 sec, DEP_1 is pushed back to GC's PROCESSING list
                 At 3 sec, ARR_1 is removed from LC's PENDING list
                 At 3 sec, ARR_1 is pushed back to LC's PROCESSING list
                This flight (DEP_1) is ahead the schedule.
                So, need to contact the Ground Controller again.
                -> scheduled time: 79.9   sysTime: 17
                At 17 sec., DEP_1 got "waitThere".
                this flight (ARR_1) is behind the schedule.
                -> scheduled time: 17   sysTime: 19
                At 19 sec., ARR_1 got "clearToLand".
                 At 33 sec, DEP_2 is removed from GC's PENDING list
                 At 33 sec, DEP_2 is pushed back to GC's PROCESSING list
                This flight (DEP_2) is ahead the schedule.
                So, need to contact the Ground Controller again.
                -> scheduled time: 186.9   sysTime: 49
...
    system clock = 7000 second
    system clock = 7200 second
< Simulation ends. >

------------------------------ SUMMARY -------------------------------
 Flight (Departure DEP_1, B727-100, Gate 1, Runway 36)
       Enters into the simulation at          : 1 sec.
       Taxiing Duration                       : 73 - 217
       Taxiing Delay                          : 2.22827
       Nominal Takeoff Time   (= NTOT)        : 186
       Sequenced Takeoff Time (= STOT)        : 268
       Actual Takeoff Time    (= ATOT)        : 289
       Runway Occupancy Time  (= ROT)         : 289 - 328
       Sequenced Delay        (= ATOT - STOT) : 21
       Runway Delay           (= ATOT - NTOT) : 103

...

 Flight (Arrival ARR_10, SF-340, Runway 36, Gate 10)
       Enters into the simulation at                  : 454 sec.
       Nominal Touchdown Time   (= NTDT)        : 468
       Sequenced TouchDown Time (= STDT)        : 998
       Actual Touchdown Time    (= ATDT)        : 991
       Runway Occupancy Time    (= ROT)         : 991 - 1018
       Sequenced Delay          (= ATDT - STDT) : 0
       Runway Delay             (= ATDT - NTDT) : 523
       Taxiing Duration                         : 1045 - 1147
       Taxiing Delay (sec.)                     : 4.52267

   TotTaxiingDelay_sec = 47.9558
   TotRunwayDelay_sec = 6486
   TotDealy_sec       = 6533.96
   AvgTaxiingDelay_sec = 2.39779
   AvgRunwayDelay_sec = 324.3
   AvgDealy_sec       = 326.698
Utilization factor (L/C) = totBusyTimeLC (=1234) / first 1 hour (= 3600) = 0.342778
Utilization factor (G/C) = totBusyTimeGC (=1188) / first 1 hour (= 3600) = 0.33
```

Figure 6.6 A Sample of the *Log.out* File.

```
Time
     FLT #        (x, y)         currCommState        permission    currMoveState       speed    accel       currLink linkLength distInLink totDist
 ---------  ------------------  ------------------   ------------   -------------     --------  --------     -------- ---------- ---------- -------

...

 320.000
     DEP_1  (4.27860, 7.23847)   readyToCommunicate     clearToTakeOff    rolling        228.557  5.65931   2006 -> 2005     347.582  322.875  8907.85
     DEP_2  (3.44770, 3.71363)   readyToCommunicate     clearToTaxi       taxiingToDepQue 27.3409  0.000000   1031 -> 2018     782.058  727.237  3832.22
     DEP_3  (3.65123, 6.51975)   readyToCommunicate     clearToTaxi       taxiingToDepQue 27.3409  0.000000   1011 -> 1014     185.988  107.714  608.929
     DEP_4  (3.24660, 6.64350)  waitControllerContact  waitThere         parking        0.000000  0.000000      0 -> 0          0  0.000000  0.000000
     DEP_5  (3.47540, 6.59730)  waitControllerContact  waitThere         parking        0.000000  0.000000      0 -> 0          0  0.000000  0.000000
     DEP_6  (3.47540, 6.59730)  waitControllerContact  waitThere         parking        0.000000  0.000000      0 -> 0          0  0.000000  0.000000
     DEP_7  (3.47540, 6.59730)  waitControllerContact  waitThere         parking        0.000000  0.000000      0 -> 0          0  0.000000  0.000000
     DEP_8  (3.47540, 6.59730)  readyToCommunicate     fileApproved      parking        0.000000  0.000000      0 -> 0          0  0.000000  0.000000
     DEP_9  (3.47540, 6.59730)  readyToCommunicate     fileApproved      parking        0.000000  0.000000      0 -> 0          0  0.000000  0.000000
    DEP_10  (3.33330, 6.32750)  readyToCommunicate     fileApproved      parking        0.000000  0.000000      0 -> 0          0  0.000000  0.000000
     ARR_1  (2.80910, 5.01500)  readyToCommunicate     clearToTaxi       parking        0.000000  0.000000      0 -> 0          0  0.000000  0.000000
     ARR_2  (2.98857, 4.53792)   readyToCommunicate     clearToTaxi       taxiingToGate   27.3409  0.000000   1029 -> 1028     233.737  4.76052  6039.35
     ARR_3  (4.13276, 8.73577)   readyToCommunicate     clearToTaxi       taxiingToGate   16.3510  4.41148   1003 -> 2003     915.958  24.2862  6123.22
     ARR_4  (4.08470, 3.21800)   waitingCommand         waitThere         onFinal        0.000000  0.000000   2021 -> 2019     493.258  0.000000  0.000000
     ARR_5  (4.08470, 3.21800)   waitControllerContact  waitThere         onFinal        0.000000  0.000000   2021 -> 2019     493.258  0.000000  0.000000
     ARR_6  (4.08470, 3.21800)   waitControllerContact  waitThere         onFinal        0.000000  0.000000   2021 -> 2019     493.258  0.000000  0.000000
     ARR_7  (4.08470, 3.21800)   readyToCommunicate     unDecided         onFinal        0.000000  0.000000   2021 -> 2019     493.258  0.000000  0.000000
     ARR_8  (4.08470, 3.21800)   readyToCommunicate     unDecided         onFinal        0.000000  0.000000   2021 -> 2019     493.258  0.000000  0.000000
     ARR_9  (4.08470, 3.21800)   readyToCommunicate     unDecided         onFinal        0.000000  0.000000   2021 -> 2019     493.258  0.000000  0.000000
    ARR_10  (4.08470, 3.21800)   readyToCommunicate     unDecided         onFinal        0.000000  0.000000   2021 -> 2019     493.258  0.000000  0.000000
321.000
     DEP_1  (4.28917, 7.45945)   readyToCommunicate     clearToTakeOff    liftingOff     234.179  0.000000   2005 -> 2002     1624.27  206.661  9139.22
     DEP_2  (3.44770, 3.68748)   readyToCommunicate     clearToTaxi       taxiingToDepQue 27.3409  0.000000   1031 -> 2018     782.058  754.578  3859.57
     DEP_3  (3.66271, 6.49626)   readyToCommunicate     clearToTaxi       taxiingToDepQue 27.3409  0.000000   1011 -> 1014     185.988  135.055  636.270
     DEP_4  (3.24660, 6.64350)  waitControllerContact  waitThere         parking        0.000000  0.000000      0 -> 0          0  0.000000  0.000000
     DEP_5  (3.47540, 6.59730)  waitControllerContact  waitThere         parking        0.000000  0.000000      0 -> 0          0  0.000000  0.000000
     DEP_6  (3.47540, 6.59730)  waitControllerContact  waitThere         parking        0.000000  0.000000      0 -> 0          0  0.000000  0.000000
     DEP_7  (3.47540, 6.59730)  waitControllerContact  waitThere         parking        0.000000  0.000000      0 -> 0          0  0.000000  0.000000
     DEP_8  (3.47540, 6.59730)  readyToCommunicate     fileApproved      parking        0.000000  0.000000      0 -> 0          0  0.000000  0.000000
     DEP_9  (3.47540, 6.59730)  readyToCommunicate     fileApproved      parking        0.000000  0.000000      0 -> 0          0  0.000000  0.000000
    DEP_10  (3.33330, 6.32750)  readyToCommunicate     fileApproved      parking        0.000000  0.000000      0 -> 0          0  0.000000  0.000000
     ARR_1  (2.80910, 5.01500)  readyToCommunicate     clearToTaxi       parking        0.000000  0.000000      0 -> 0          0  0.000000  0.000000
     ARR_2  (2.96257, 4.53519)   readyToCommunicate     clearToTaxi       taxiingToGate   27.3409  0.000000   1029 -> 1028     233.737  32.1014  6066.69
     ARR_3  (4.12726, 8.71929)   readyToCommunicate     clearToTaxi       taxiingToGate   19.9686  3.61763   1003 -> 2003     915.958  42.4460  6141.38
     ARR_4  (4.08470, 3.21800)   receivingCommand       waitThere         onFinal        0.000000  0.000000   2021 -> 2019     493.258  0.000000  0.000000
     ARR_5  (4.08470, 3.21800)   waitControllerContact  waitThere         onFinal        0.000000  0.000000   2021 -> 2019     493.258  0.000000  0.000000
     ARR_6  (4.08470, 3.21800)   waitControllerContact  waitThere         onFinal        0.000000  0.000000   2021 -> 2019     493.258  0.000000  0.000000
     ARR_7  (4.08470, 3.21800)   readyToCommunicate     unDecided         onFinal        0.000000  0.000000   2021 -> 2019     493.258  0.000000  0.000000
     ARR_8  (4.08470, 3.21800)   readyToCommunicate     unDecided         onFinal        0.000000  0.000000   2021 -> 2019     493.258  0.000000  0.000000
     ARR_9  (4.08470, 3.21800)   readyToCommunicate     unDecided         onFinal        0.000000  0.000000   2021 -> 2019     493.258  0.000000  0.000000
    ARR_10  (4.08470, 3.21800)   readyToCommunicate     unDecided         onFinal        0.000000  0.000000   2021 -> 2019     493.258  0.000000  0.000000
```

Figure 6.7 A Sample of the *flightState.out* File.

```
ARR_1(system clock: 65)
Static  Path: 1003 -> 2003 -> 1005 -> 1008 -> 1010 -> 1012 -> 1015 -> 1018 -> 1021 -> 1020 -> 1024 -> 1026 -> 1030 -> 1029 -> 1028 -> 12
td_SP_2 Path: 1003 -> 2003 -> 1005 -> 1008 -> 1010 -> 1012 -> 1015 -> 1018 -> 1021 -> 1020 -> 1024 -> 1026 -> 1030 -> 1029 -> 1028 -> 12

DEP_1(system clock: 73)
Static  Path: 1009 -> 1011 -> 1014 -> 1016 -> 1017 -> 1019 -> 1020 -> 1024 -> 1027 -> 1031 -> 2018 -> 2021
td_SP_2 Path: 1009 -> 1011 -> 1014 -> 1016 -> 1017 -> 1019 -> 1020 -> 1024 -> 1027 -> 1031 -> 2018 -> 2021

DEP_2(system clock: 180)
Static  Path: 1009 -> 1011 -> 1014 -> 1016 -> 1017 -> 1019 -> 1020 -> 1024 -> 1027 -> 1031 -> 2018 -> 2021
td_SP_2 Path: 1009 -> 1010 -> 1012 -> 1015 -> 1018 -> 1021 -> 1022 -> 1025 -> 1027 -> 1031 -> 2018 -> 2021

ARR_2(system clock: 192)
Static  Path: 1015 -> 1018 -> 1021 -> 1020 -> 1024 -> 1026 -> 1030 -> 1029 -> 1028 -> 12
td_SP_2 Path: 1015 -> 1018 -> 1021 -> 1020 -> 1024 -> 1026 -> 1030 -> 1029 -> 1028 -> 12

ARR_3(system clock: 291)
Static  Path: 1003 -> 2003 -> 1005 -> 1008 -> 1010 -> 1012 -> 1015 -> 1018 -> 1021 -> 1020 -> 1024 -> 1026 -> 1030 -> 1029 -> 1028 -> 12
td_SP_2 Path: 1003 -> 2003 -> 1005 -> 1008 -> 1010 -> 1012 -> 1015 -> 1018 -> 1021 -> 1020 -> 1024 -> 1026 -> 1030 -> 1029 -> 1028 -> 12


...


DEP_8(system clock: 908)
Static  Path: 1011 -> 1014 -> 1016 -> 1017 -> 1019 -> 1020 -> 1024 -> 1027 -> 1031 -> 2018 -> 2021
td_SP_2 Path: 1011 -> 1014 -> 1012 -> 1015 -> 1018 -> 1021 -> 1022 -> 1025 -> 1027 -> 1031 -> 2018 -> 2021

DEP_9(system clock: 968)
Static  Path: 1011 -> 1014 -> 1016 -> 1017 -> 1019 -> 1020 -> 1024 -> 1027 -> 1031 -> 2018 -> 2021
td_SP_2 Path: 1011 -> 1014 -> 1016 -> 1017 -> 1019 -> 1020 -> 1024 -> 1027 -> 1031 -> 1032 -> 1033 -> 2018 -> 2021
```

Figure 6.8 A Sample of *TaxiPath.out* File.

6.5 Computational Results

6.5.1 Total Taxiing Time as Related to Various Network Assignment Strategies

Table 6.4 shows results for the static and time-dependent NAP in total taxiing time. As shown in the table, the impact of the time-dependent NAP varies depending on a predefined unit delay time, tentatively called Conflict Delay Time (CDT), which represents the impact of a conflicting flight on taxiing delay and is used when the time-dependent taxiing route is planned. For example, if CDT is set to 5 seconds, then the travel time for a certain link increases 5 seconds for every one possible conflicting aircraft.

Table 6.4 Total Taxiing Times.

(Unit: Seconds)

Scenarios			Number of operations per hour								
Network Assign.	Sequence	Comm.	10	15	20	25	30	35	40	45	50
Static Network Assign.	Pure FCFS	Voice Ch.	1650.0	2410.8	3122.6	3917.4	4570.9	5265.3	5886.8	6584.4	7141.3
		Data Link	1648.1	2410.1	3121.8	3916.7	4574.8	5258.2	5889.1	6584.3	7150.6
	FCFS w/ Land.	Voice Ch. **(Base Sce.)**	1650.4	2413.2	3122.9	3916.6	4577.0	5270.9	5888.2	6589.8	7140.8
		Data Link	1647.9	2409.5	3120.3	3913.8	4573.2	5259.1	5885.9	6577.9	7144.2
	2-exch.	Voice Ch.	1650.6	2413.2	3123.0	3917.6	4577.8	5271.4	5889.7	6594.3	7142.1
		Data Link	1648.1	2409.6	3120.2	3913.1	4573.0	5262.6	5885.8	6580.8	7148.0
Time-Dep. Network Assign. (CDT* = 1 sec.)	Pure FCFS	Voice Ch.	1650.0	2410.8	3122.6	3917.4	4570.9	5265.3	5886.8	6584.4	7141.3
		Data Link	1648.1	2410.1	3121.8	3916.7	4574.8	5258.2	5889.1	6584.3	7150.6
	FCFS w/ Land.	Voice Ch.	1650.4	2413.2	3122.9	3916.6	4577.0	5270.9	5888.2	6589.8	7140.8
		Data Link	1647.9	2409.5	3120.3	3913.8	4573.2	5259.1	5885.9	6577.9	7144.2
	2-exch.	Voice Ch.	1650.6	2413.2	3123.0	3917.6	4577.8	5271.4	5889.7	6594.3	7142.1
		Data Link	1648.1	2409.6	3120.2	3913.1	4573.0	5262.6	5885.8	6580.8	7148.0
Time-Dep. Network Assign (CDT = 2 sec.)	Pure FCFS	Voice Ch.	1650.0	2410.8	3123.7	3917.4	4570.9	5265.3	5886.6	6587.7	7142.8
		Data Link	1648.1	2410.1	3121.8	3916.7	4574.8	5258.2	5890.1	6584.8	7153.2
	FCFS w/ Land.	Voice Ch.	1650.4	2413.2	3122.9	3916.6	4577.0	**5270.0**	5889.2	6589.8	7141.9
		Data Link	1647.9	2409.5	3120.8	3914.2	4573.6	5259.9	5886.5	6577.9	7145.4
	2-exch.	Voice Ch.	1650.6	2413.2	3123.0	3918.1	4577.8	5271.4	5890.7	6593.6	7142.4
		Data Link	1648.1	2409.6	3120.7	3913.1	4573.4	5262.6	5886.4	6580.8	7148.4
Time-Dep. Network Assign. (CDT = 5 sec.)	Pure FCFS	Voice Ch.	1650.8	2410.8	3125.0	3920.5	4575.6	5266.3	5890.5	6597.3	7150.0
		Data Link	1648.1	2410.5	3122.3	3919.0	4580.5	5266.8	5895.5	6593.2	7161.9
	FCFS w/ Land.	Voice Ch.	1651.3	2413.6	3124.8	3918.7	4578.2	5273.9	5900.2	6598.9	7151.7
		Data Link	1647.9	2411.3	3122.5	3916.6	4579.0	5268.2	5895.1	6589.0	7154.5
	2-exch.	Voice Ch.	1651.5	2413.6	3125.5	3919.8	4580.5	5274.6	5899.0	6602.3	7156.7
		Data Link	1648.1	2411.4	3122.5	3916.4	4579.2	5269.3	5893.6	6592.7	7158.1

*CDT: Conflict Delay Time.

The simulation results summarized in the Table 6.4 indicate that the time-dependent taxiing routes planed based on 5 seconds of CDT cause more taxiing time than the static taxiing routes. In the case that CDT is 2 seconds, some of time-dependent taxiing plans help to reduce the taxiing time but some of them still cause more taxiing time than static taxiing plans. In the case of 1 second of CDT, the resulting taxiing times are exactly the same ones as those founded in the static case. Based on these results, we can postulate that CDT value for this case study will be between 1.0 and 2.0. (For the computational convenience, 2.0 of CDT value will be used from this point.) We also can see that the communication method does not show any correlation with taxiing time itself.

6.5.2 Average Runway Delay for Analysis of Aircraft Sequencing Strategies

The average runway delays for all three types of sequencing strategies are summarized in Table 6.5. The results clearly show that, if the 2-exchange sequences are used in runway operations, the savings in runway delays reaches up to 15% compared to base scenario. And it is also showed that the communication based on the data link help in saving the runway delay.

Table 6.5 Average Runway Delays resulting from the Simulation.

(Unit: Seconds / Flight)

Scenarios			Number of operations per hour								
Network Assign.	Sequence	Comm.	10	15	20	25	30	35	40	45	50
Static Network Assign.	Pure FCFS	Voice Ch.	17.9	31.5	36.0	45.9	48.0	100.4	102.8	170.6	324.4
		Data Link	8.8	19.3	24.3	28.6	34.2	64.8	77.1	112.7	211.6
	FCFS w/ Land.	Voice Ch. (**Base Sce**.)	18.4	30.2	32.8	40.9	45.5	63.5	81.6	94.8	137.6
		Data Link	10.7	21.9	23.6	32.3	35.7	52.1	69.0	83.0	123.6
	SWAP	Voice Ch.	17.7	29.7	32.9	39.4	44.8	62.4	76.6	90.1	127.6
		Data Link	9.7	21.0	22.4	27.9	33.8	47.9	60.7	76.2	110.1
Time-Dep. Network Assign. (5sec)	Pure FCFS	Voice Ch.	17.9	31.5	36.1	45.9	48.0	100.4	102.8	170.7	324.4
		Data Link	8.8	19.3	24.3	28.6	34.2	64.8	77.1	112.7	211.6
	FCFS w/ Land.	Voice Ch.	18.4	30.2	32.8	40.9	45.5	63.5	81.7	94.8	137.7
		Data Link	10.7	21.9	23.6	32.3	35.7	52.1	69.0	83.0	123.6
	SWAP	Voice Ch.	17.7	29.7	32.9	39.4	44.8	62.4	76.7	90.5	127.6
		Data Link	9.7	21.0	22.4	27.9	33.8	47.9	60.7	76.2	110.1

It should be pointed out that average delays obtained from the simulation model are not same as those from the analytical solution shown in Table 6.6. This is because in some cases, analytical sequences cannot be implemented due to the communication time lags (delays) as well as the several assumptions used in simulation model.

Table 6.6 Average Runway Delays resulting from Sequencing Step.

(Unit: Seconds / Flights)

Sequence	Number of operations per hour								
	10	15	20	25	30	35	40	45	50
FCFS W/ Land. (1)	10.9	23.2	24.9	35.1	38.3	56.3	73.2	89.0	130.5
2-exch. (2)	9.6	22.0	23.0	29.6	35.7	51.5	64.0	81.1	115.6
Savings (= (1)-(2))	1.3	1.2	1.9	5.5	2.6	4.8	9.2	7.9	14.9

The average delays obtained using the sequencing methods are illustrated in Figures 6.9 and 6.10. Figure 6.9 indicates that, if the runway practical capacity is decided at the level of four minutes of delay per aircraft, the runway capacity is about 46 or 47 arrivals per hour. This seems to correlate well with the analytical results obtained using the airport capacity model (ACM) for the same aircraft mix.

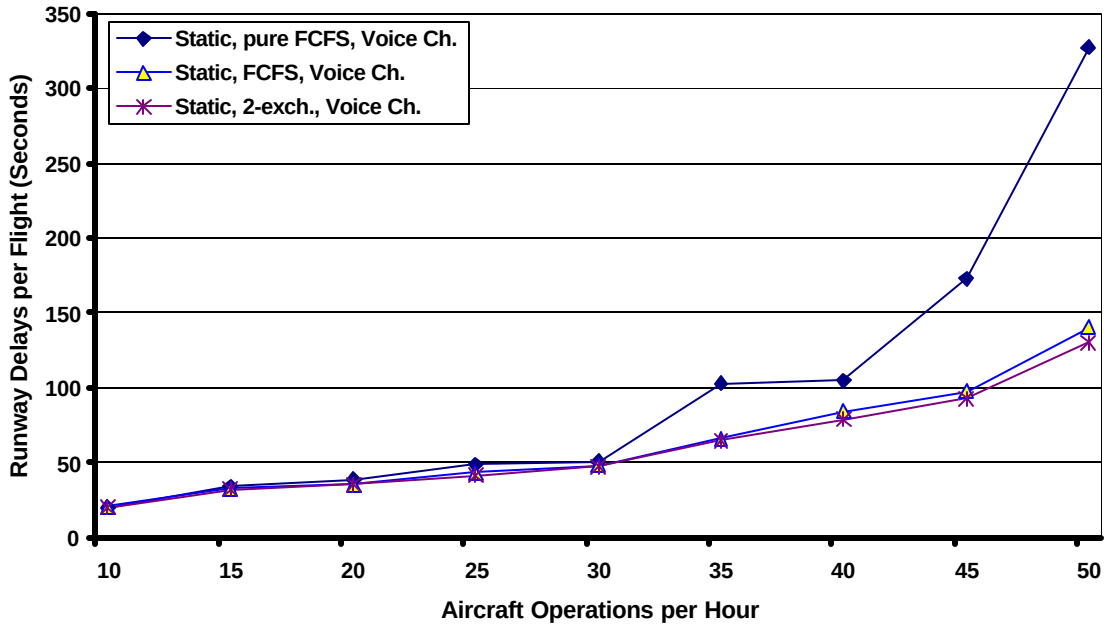


Figure 6.9 Average Runway Delay by Sequencing Methods (Communication: Voice Channel).

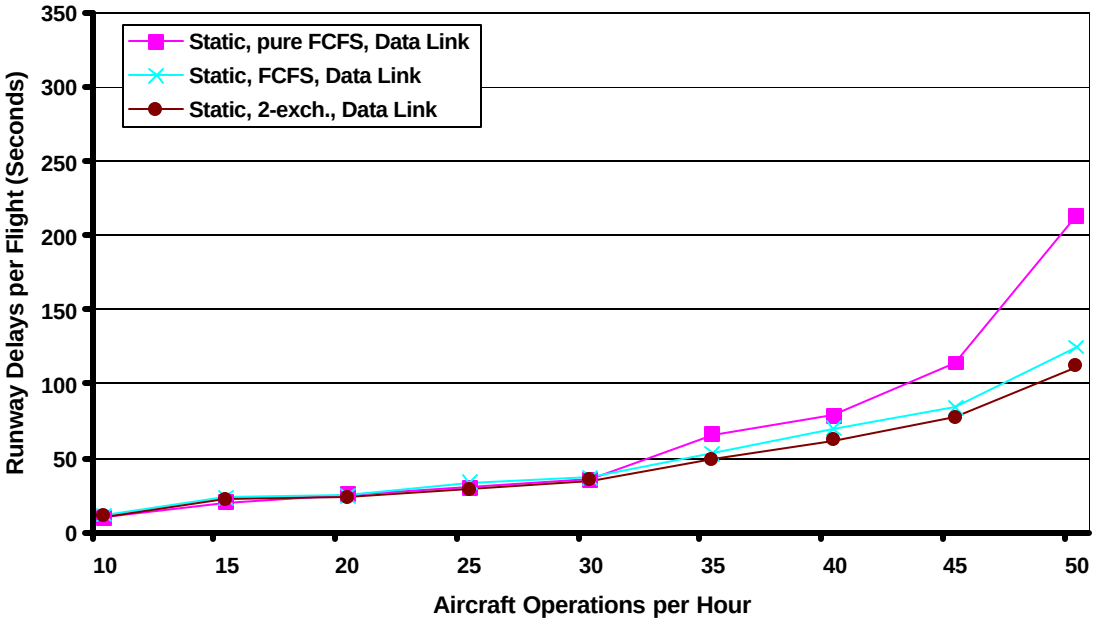


Figure 6.10 Average Runway Delay by Sequencing Methods (Communication: Data Link).

6.5.3 Total Operation Time

Total operation time is a good index to compare various air traffic control strategies. The total operation obtained for all 12 scenarios are shown in Table 6.7. The results indicate that the total operation time can be reduced by up to 9.2% if all three advanced air traffic control technologies, including 2-excahnge sequencing, time-dependent network assignment technique and data link are used. It is also evident that among these three technologies, the pilot-controller communication method is the most effective at reducing total operation time.

Interestingly, the time-dependent taxiing planning does not provide a significant advantages at reducing delays. Nevertheless, it should be noted that time-dependent taxiing feature in the simulation model is inevitable to model the proper temporal changes in the operation of every taxiway link. Furthermore, the time-dependent taxiing planning algorithm is expected to play a substantive role to reduce the total operation time at airports having more complicated taxiway network with higher taxiing demands.

The results of three sequencing strategies are compared in Figure 6.11. As expected, the pure FCFS policy yields the highest delays. The impact of two communication methods on the total delays is compared in Figure 6.12. Here we note certain reduction in delays when pilot controller datalink communications are used.

Table 6.7 Total Operation Time [1)].

(Unit: Seconds)

Scenarios			Number of operations per hour								
Network Assign.	Sequence	Comm.	10	15	20	25	30	35	40	45	50
Static Network Assign.	Pure FCFS	Voice Ch.	2589.1	4017.4	5345.2	6935.3	8272.7	11421.1	13029.4	17664.3	27150.2
		Data Link	2420.3	3712.9	4971.1	6328.2	7656.7	9952.9	11756.1	14794.7	21239.0
	FCFS	Voice Ch. (**Base Sce.**)	2593.9	3998.9	5277.3	6814.0	8198.2	10136.0	12171.6	14264.9	17810.1
		Data Link	2435.4	3743.1	4949.0	6406.0	7691.6	9492.4	11409.6	13427.4	16821.8
	2-exch.	Voice Ch.	2587.3	3989.7	5279.3	6774.8	8175.7	10102.2	11974.1	14056.6	17323.6
		Data Link	2426.2	3731.1	4927.1	6303.7	7635.8	9353.1	11087.6	13128.2	16155.5
Time-Dep. Network Assign.	Pure FCFS	Voice Ch.	2589.1	4017.4	5346.9	6935.3	8272.7	11421.1	13028.6	17665.5	27150.2
		Data Link	2420.3	3712.9	4971.1	6328.2	7656.7	9952.9	11756.5	14794.7	21239.1
	FCFS	Voice Ch.	2593.9	3998.9	5277.3	6813.9	8198.2	10135.2	12175.4	14263.1	17815.9
		Data Link	2435.4	3743.1	4949.0	6406.0	7691.8	9492.4	11410.9	13427.4	16821.8
	2-exch.	Voice Ch.	2587.3	3989.7	5279.3	6774.4	8175.7	10102.2	11977.9	14076.4	17323.9
		Data Link	2426.2	3731.1	4927.1	6303.7	7636.0	9353.1	11088.9	13128.2	16155.7

[1)] Total Operation Time = $\sum_{i = \text{all flights}}$ (Time to finish operation$_i$ - Time to enter the system$_i$).

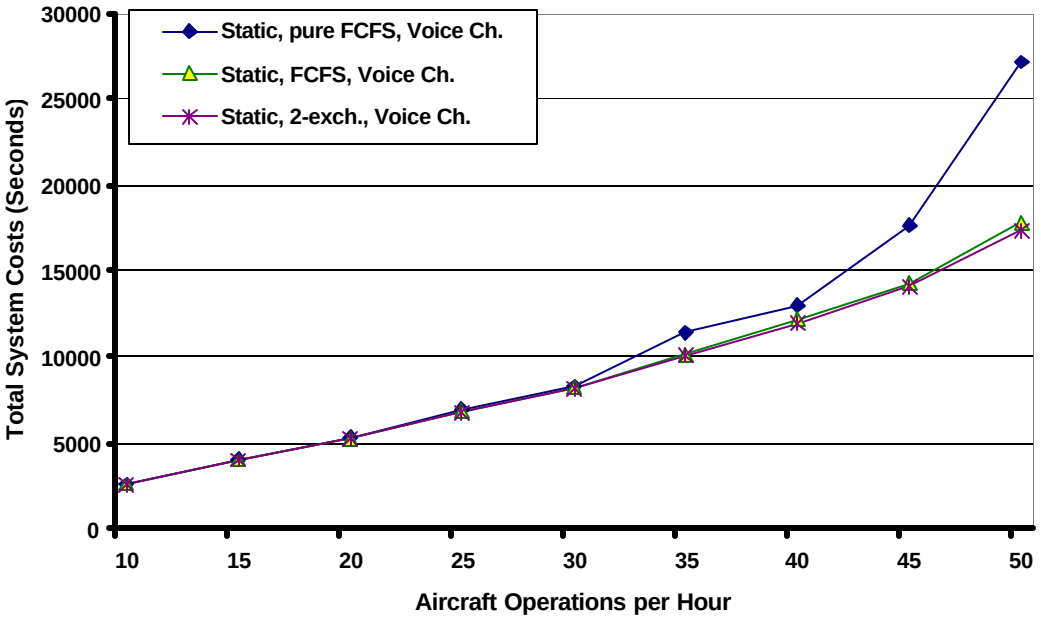


Figure 6.11 Total System Costs by Sequencing Methods (from the Simulation Run).

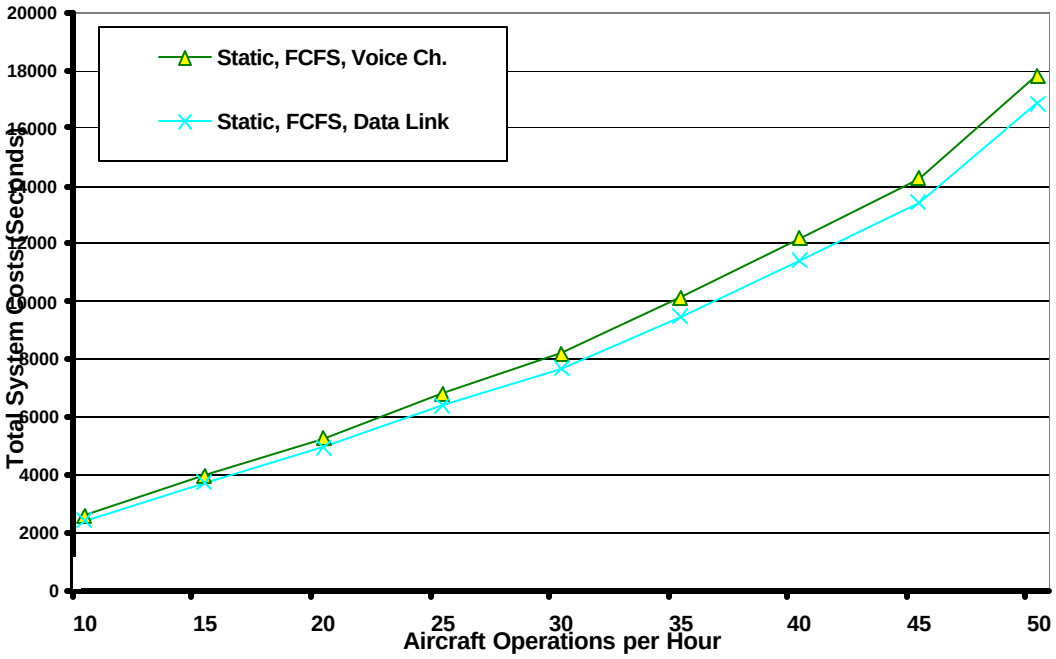


Figure 6.12 Total System Costs by Sequencing Methods (from the Simulation Run).

Chapter 7. Summary, Conclusions and Future Research

7.1 Summary of Results

Unless the weather is adverse, flight delays occur mainly due to excessive demands on airport facilities such as runways, taxiways, gates, communication system, etc. This research focuses on the development of optimization models and algorithms and a computer simulation model to study various activities inside the airport terminal area. These models are intended to help reduce congestion on runways and taxiways.

In order to manage aircraft traffic at busy airports and terminal areas, aircraft sequencing methods can be used in conjunction with advanced Air Traffic Control (ATC) automation tools. This research discusses a combinatorial optimization approach to the Aircraft Sequencing Problem (ASP). Consideration is given to aircraft arrival and departure streams on a single runway or in closely spaced runways where arrivals and departures are dependent. Previous studies conducted in this area have mainly focused on the sequencing problem for arriving aircraft. Since many airports in the U.S. have dependent arrival and departure stream operations, consideration is given to both conditions simultaneously. This makes the problem more realistic yet at the same time, more challenging. Starting with preliminary information about the aircraft such as the aircraft types, desired arrival/departure times, maximum delay times, minimum separation rules, etc., the problem can be formulated as a traveling salesman problem with time-windows, and with nonconsecutive separation enforcement.

To solve the ASP, an exact approach using integer programming techniques is developed. Tight lower bounds are generated using the Reformulation-Linearization Technique (RLT) of Sherali and Adams (1990, 1994). Compared with the linear programming relaxation of the original model, the proposed reformulated ASP problems provide tighter representations, producing significantly improved lower bounds. As a result, several previously unsolvable instances are now computationally tractable within the set limits. The computational results show that, by changing the traditional FCFS sequence intelligently, about 17% of the overall operational time can be saved.

An important consideration for the ASP algorithm is its eventual implementation in real-time runway operations. As such, the computation time required to solve the ASP is of paramount importance. To reduce computation time, we suggest several heuristic approaches by modifying 2-exchange and swap methods which are widely used in solving TSP problems. Here, the exact method serves to help enumerate the heuristic procedures for reasonably sized problems as well as to develop such heuristic procedures themselves. Computational results show that the 2-exchage heuristic method provides sequences very close to exact solutions in reasonable time.

Given an optimal sequence output from the ASP model, the Network Assignment Problem (NAP) on the taxiway-runway system is considered in a single framework. To resolve the relationships between runway and taxiway operations, it is desirable to establish certain connections between ASP and NAP. In our case, we place a higher priority on runway operations. This means that an optimal aircraft sequence for runway operations which completes the scheduled departures and landings as soon as possible will be decided first. Taking this sequence as a given condition, the NAP problem is solved. In this reaserch, the quasi-dynamic network assignment stategy is adopted as the method for NAP. By taking advantage of the dynamic aspects of the problem, potential conflicts on the taxiways are detected and avoided if needed. This feature helps design a more effective network assignment procedure.

To evaluate the system performance in detail, a microscopic simulation model has been developed. Some salient points of the developed model include: 1) a hybrid-type simulation model that can analyze the total delay due not only to network congestion but also to communication channel capacity; 2) an aircraft-following model that captures more dynamic and detailed behavior of the moving aircraft, and 3) computer codes are developed in ANSI C++ for the sake of portability. Furthermore, the object-oriented-programming concept employed in developing simulation model is expected to render the source codes more readable and handy to modify for the future development.

Adopting Washington National (DCA) Airport as a sample airport, a case study is presented along with computational results. Results confirm that for high demand levels, communication can be another source of delay that is not considered in the ASP and NAP problems. Results show that if an advanced ATC system using the 2-exchange sequence in runway operations and the data-link system in communication is used, the savings in runway delays reaches up to 19% compared to the base scenario. Results also reveal that up to 5% of the total operational time in

the terminal area can be reduced by upgrading the current voice-channel communication system to the data-link communication system.

7.2 Recommendations for Future Research

Applying Lagrangian relaxation techniques to ASP: Lagrangian relaxation can be applied to solving integer programming problems exactly or approximately in a more effective fashion by exploiting inherent special structures. Lagrangian relaxation can also be applied in conjunction with RLT, by using it to efficiently solve the LP relaxations through duality considerations.

Consideration of gate allocation: It is frequently observed at busy airports that some landing flights wait on the taxiway for gates to be vacated. Gate delay is another major source of total delay which has not been considered in this research. Gates can be managed more efficiently by solving a so-called "Gate Assignment Problem (GAP)" (see Sherali and Brown, 1994). Since gates are facilities for both aircraft and passengers, the passenger delays are also considered in GAP. The effect of GAP could be incorporated in future extensions of our research.

Modifying the shortest path algorithm in the time-dependent NAP: Waiting times at the nodes can be considered when the dynamic shortest paths are calculated. Instead of detouring the blocked link, the aircraft can wait at the crossing point until the blockage is released. This is unlike the ground transportation network where any waiting at intersections is prohibited due to the operational characteristics of highways. To take node waiting into account, Orda and Rom's study (1990) discussed in Chapter 4 can be applied.

Consideration of stochastic factors in the simulation model: For a more realistic analysis, probabilistic functions can be used to characterize the following possible stochastic components: 1) the time duration for each phase of the communication process such as receiving request, judging, sending commands, receiving confirmations, and communication processes; 2) aircraft performance on the runway with modifications to the descent flight path, the threshold crossing height, the threshold crossing speed, braking speed, etc., and 3) the lift-off speed in the departure procedure. In order to obtain these probability functions, a data collection process and statistical analyses could be conducted in the future. (All aspects of the second point above have already been studied in detail at Virginia Tech.)

Enhancing certain features of the simulation model: It is assumed in the simulation model that only a single runway is used for both arrival and departure operations. For the simulation model to be more versatile, it needs to be extended to handle a more diverse set of runway configurations. These include multiple runways with dependent operational conditions along with runway crossing procedures. In addition, there are several important features that have not been implemented in this research: 1) weather condition, 2) multiple controllers, 3) pushback behavior in association with gate occupancy, 4) service vehicle movements, 5) icing process, 6) holding stack, etc. All these are valid extensions to this model.

Data and problem horizon: From a practical implementation point of view, it is desirable to divide the entire data horizon into several relatively smaller problem horizons. The fragmentation of the data horizon provides us with two practical benefits. First, we can handle the data in a reasonable computational time, which is a critical factor in real-time ATC systems. For example, as discussed earlier, the computational time for ASP grows dramatically as the number of aircraft considered increases. Second, by making the problem horizon smaller, we can both reduce the uncertainty and re-consider residual (or missed) operations in the next problem.

Using this approach, operations that have been considered during some imminent duration periods are eliminated from further consideration (see Figure 7.1). The operations starting at the end of the imminent duration period constitute new operations for the next problem, having suitably revised input data. On the other hand, aircraft that are at the initial or intermediate stages of operations will continue according to the prescribed related decisions as determined in the previous periods' problems. Figure 7.2 displays the problem structure and data- flows for the ASP and NAP models in a multi-data-horizon framework.

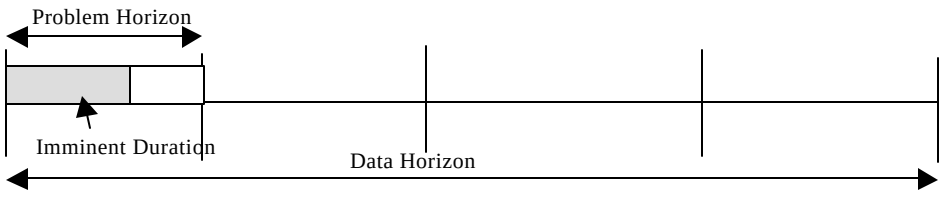


Figure 7.1 Definition of Problem and Data Horizons.

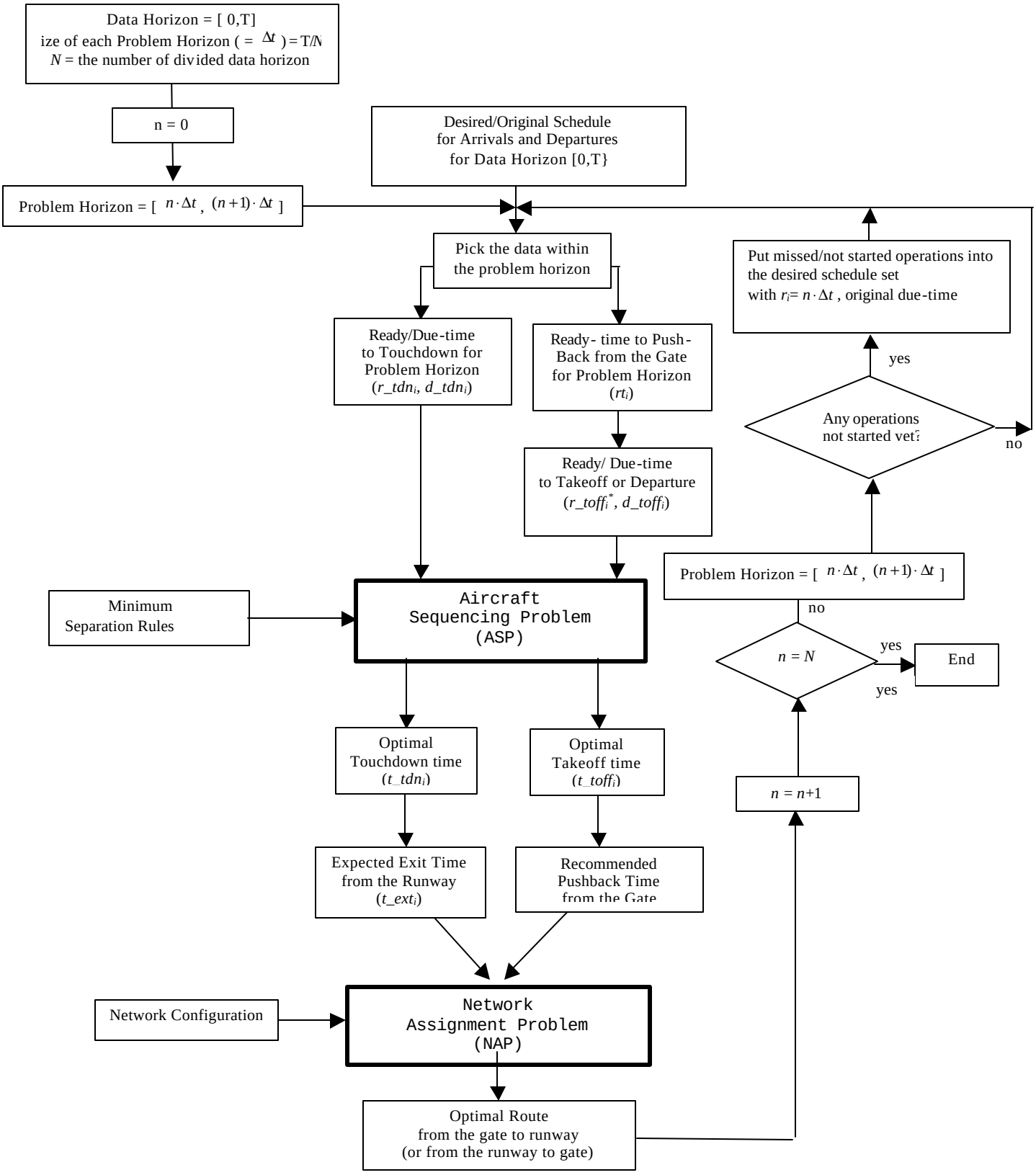


Figure 7.2 Problem Structure and Data-Flow for the ASP and NAP Model Framework (Multi-Data Horizon).

Bibliography

Adams, W. P., and H. D. Sherali. A tight linearization and an algorithm for zero-one quadratic programs. *Management Science*, 32:1274-1290, 1986.

Adams, W. P., and H. D. Sherali. Mixed-integer bilinear programming problems. *Mathematical Programming*, 59(3):279-305, 1993.

Ahuja, R. K., T. L. Magnanti, and J. B. Orlin. *Network Flows*. Prentice Hall, New Jersey, 1993.

Andreatta, G., and G. Romanin-Jacur. Aircraft flow management under congestion. *Transportation Science*, 21(4):249-253, 1987.

Ascheuer, N., M. Fischetti, and M. Grotschel. Solving the asymmetric traveling salesman problem with time-window by branch-and-cut. Preprint SC-99-31, ZIB Berlin, Germany, 1999.

Baker, K. R. *Introduction to Sequencing and Scheduling*. John Wiley & Sons, Inc., New York, 1974.

Baker, E. K., and J. R. Schaffer. Solution improvement heuristics for the vehicle routing and scheduling problem with time-window constraints. *American Journal of Mathematical and Management Sciences* 6(3):261-300, 1986.

Baker, K. R., and L. E. Schrage. Finding an optimal sequence by dynamic programming: An extension to precedence-related tasks. *Operations Research*, 26(1): , 1978.

Bakker, G., H. Enting, and K. Nienwenhuys. OMT object model, Method Engineering Encyclopaedia. Online. Available: http://www.univ-paris1.fr/CRINFO/dmrg/MEE/ misop009, Dec. 1995.

Bazaraa, M. S., J. J. Jarvis, and H. D. Sherali. *Linear Programming and Network Flows*. Second Edition, John Wiley and Sons, New York, 1990.

Bellman, R. On a routing problem. *Quarterly of Applied Mathematics*, 16:87-90, 1958.

Bianco, L., G. Rinaldi, and A. Sassano. A combinatorial optimization approach to aircraft sequencing problem. *Flow Control of Congested Networks*, Edited by A. R. Odoni et al., NATO ASI series, F38:323-339, 1987.

Bianco, L., S. Ricciardelli, G. Rinaldi, and A. Sassano. Scheduling tasks with sequence-dependent processing times. *Naval Research Logistics*, 35:177-184, 1988.

Bianco, L., P. Dell'Olmo, and S. Giordani. Scheduling models and algorithms for TMA traffic management. *Modeling and Simulation in Air Traffic Management*, Edited by Bianco, L. et al., Springer, 139-167, 1997.

Bodin, L., B. Golden, A. Assad, and M. Ball. Routing and scheduling of vehicles and crews- The state-of-the-art. *Computers & Operations Research*, 10(2):63-211, 1983.

Booch, G. *Object-oriented Analysis and Design with Application*. Addison-Wesley, 1994.

Booch, G., J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Massachusetts, 1998.

Clarke, G., and J. Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research*, 12:568-581, 1964.

Cook, K. L., and E. Halsey. The shortest route through a network with time-dependent internodal transit time. *Journal of Mathematical Analysis and Application*, 14:493-498, 1966.

Dear, R. D. The dynamic scheduling of aircraft in the near terminal area. *MIT Flight Transportation Laboratory Report* R76-9, MIT, Cambridge, MA., 1976.

Dear, R. G., and Y. S. Sherif. An algorithm for computer assisted sequencing and scheduling of terminal area operations. *Transportation Research*, 25A:129-139, 1991

Derr, K. W. *Applying OMT*: *A Practical Step-by-step Guide to Using the Object Modeling Technique*. SIGS Books, New York, 1995.

Desrochers, M., J. Desrosiers, and M. Solomon. A new optimization algorithm for the vehicle routing problem with time-windows. *Operations Research* 40(2):342-354, 1992.

Desrosiers, J., Y. Dumas, and M. Solomon. A dynamic programming solution of the large-scale single-vehicle dial-a-ride problem with time-windows. *American Journal of Mathematical and Management Sciences*, 6(3):301-325, 1986.

Dial, R., F. Glover, D. Karney, and D. Klingman. A Computational analysis of alternative and labeling techniques for finding shortest path trees. *Networks*, 9:215-245, 1979.

Dijkstra, E. W. A note on two Problems in connection with Graphs. *Numerische Mathematik*, 1:269-271, 1959.

Dreyfus, S. E. An appraisal of some shortest path algorithms. *Operations Research*, 17:395-412, 1969.

Driscoll, P. J. A new hierarchy of relaxations for 0-1 mixed integer problems with application to some specially structured problem. Ph. D. Dissertation, Virginia Polytechnic Institute and State university, 1995.

Dumas, Y., J. Desrosiers, and F. Soumis. The pickup and delivery problem with time-windows. *European Journal of Operational Research*, 54:7-22, 1991.

Dumas, Y., J. Desrosiers, E. Gelinas, and M. M. Solomon. An optimal algorithm for the traveling salesman problem with time-windows. *Operations Research*, 43(2):367-371, 1995.

Frank R. L., and P. Wolfe. An algorithm of Quadratic Programming. *Naval Research Logistics*, 3:95-110, 1956.

French, S. *Sequencing and Scheduling: An Introduction to the Mathematics of the Job Shop*. John Wiley & Sons, Inc., New York, 1982.

Friesz, T. L., J. Luque, R. L. Tobin, and B. Wie. Dynamic network traffic assignment considered as a continuous time optimal control problem. *Operations Research*, 37(6):893-901, 1989.

Friesz, T. L., D. Bernstein, T. E. Smith, R. L. Tobin, and B. Wie. A variational inequality formulation of the dynamic network user-equilibrium problem. *Operations Research*, 41(1):179-191, 1993.

Gallo, G., and S. Pallottino. Shortest path methods in transportation models. *Transportation Planning Models*, Ed. M. Florian. Elsevier Science Publishers B. V., North-Holland, 227-256.

Gallo, G., and S. Pallottino. Shortest path algorithms. *Annals of Operations Research*, 13:3-79, 1988.

Garey, M., and D. Johnson. *Computers and Intractability*: *A Guide to the Theory of NP-completeness*. Freeman, San Francisco, California, 1979.

Gendreu, M., A. Hertz, and G. Laporte. New insertion and postoptimization procedures for the traveling salesman problem. *Operations Research*, 40(16):1086-1093, 1992.

Gendreu, M., A. Hertz, G. Laporte and M. Stan. A generalized insertion heuristic for the traveling salesman problem with time-windows. *Operations Research*, 46(3):330-335, 1998.

Gerlough, D. L., and M. J. Huber. *Traffic flow theory*. TRB Special report 165, TRB, 1975.

Golden, B., T. Doyle, and W. Stewart, Jr. Approximate traveling salesman algorithms. *Operations Research*, 28(3):694-711, 1980.

Golden, B., and W. R. Stewart. Empirical analysis of heuristics. *The Traveling Salesman Problem.* Lawler, E. L., J. L. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys (Ed.), John Wiley & Sons, New York, 207-249, 1985.

Glover, F., D. Klingman, and N. Phillips. A new ploynomially bounded shortest path Algorithm. *Operations Research*, 33(1):65-73, 1985.

Halpern, J. L. Shortest route with time-dependent length of edges and limited delay possibilities in Nodes. *Zeitschrift fur Operations Research*, 21:177-124, 1977.

Held, M. and R. M. Karp. A dynamic approach to sequencing Problems. *Journal of Society of Applied Mathematics*, 10(1): , 1962.

Jacobson, I. *Object-oriented software engineering: A use case driven approach*. Addison-Wesley, Massachusetts, 1992.

Janson, B. N. A convergent algorithm for urban road networks. *Transportation Research Record*, 1318:69-80, 1991.

Kafura D. *Object-oriented Software Design and Construction with C++*. Prentice-Hall, Inc. New Jersey, 1998.

Kanellakis, P., and C. H. Papadimitriou. Local search for the asymmetric traveling salesman problem. *Operations Research*, 28(5):1086-1099, 1980.

Kaufman, D. E., and R. L. Smith. Fastest paths in time-dependent networks for IVHS Application. *IVHS Journal*, 1:1-11, 1993.

Kim, B. J., A. A. Trani, X. Gu, and C. Zhong. Computer simulation model for airplane landing-performance prediction. *Transportation Research Record*, 1562:53-62, 1996.

Laporte G. The traveling salesman problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59:231-247, 1992a.

Laporte G. The vehicle routing problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59:345-358, 1992b.

Law, A. M. and W. D. Kelton. *Simulation modeling and analysis*, 2nd Edition. McGraw-Hill, Inc., New York, 1991.

Lawler, E. L. *Combinatorial Optimization*: *Networks and Matroids*. Holt, Rinehart and Winston, New York, 1976.

Lawler, E. L., J. L. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. The Traveling Salesman Problem. John Wiley & Sons, New York, 1985.

Leonard, D. R., J. B. Tough, and P. C. Baguley. CONTRAM - A traffic assignment model for predicting flows and queues during peak periods. TRRL SR 568, 1978.

Lieberman, E. and A. K. Rathi. *Traffic Flow Theory, A State-of-the-Art Report*. TRB, 1992. Available: http://www.tfhrc.gov/its/tft/chap10.pdf.

Lin, S. Computer solutions to the traveling salesman problem. *Bell System Technical Journal*, 44:2245-2269, 1965.

Lin, S., and B. W. Kernighan. An effective heuristic algorithm for the traveling salesman problem. *Operations Research*, 21:498-516, 1973.

Luffsey, W. S. *Air traffic control*: *How to become an FAA air traffic controller*. Random House, New York, 1990.

May, A. D. *Traffic flow fundamentals*. Prentice-Hall, Inc., New Jersey, 1990.

Mahmassani, H. S. Dynamic traffic simulation and assignment: Models, algorithms and application to ATIS/ATMS evaluation and operation. *Operation research and decision aid methodologies in traffic and transportation management*, Edited by Martine L. et al., NATO ASI Series, F166:104-135, 1998.

Morton, T. E., and D. W. Pentico. *Heuristic Scheduling Systems*: *with applications to the production systems and project management*, John Wiley & Sons, Inc., New York, 1993.

Neuman, F., and H. Erberger. Analysis of sequencing and scheduling methods for arrival traffic. *NASA Technical Memorandum* 102795, Apr. 1990.

Newell, G. F. Airport capacity and delay. *Transportation Science*, 13(3):201-241, 1979.

Odoni , A. R. The flow management problem in air traffic management. *Flow control of congested networks*, Edited by A. R. Odoni et al., NATO ASI series, F38:269-288, 1987.

Or, I. Traveling salesman–type combinatorial problems and their relation to the logistics of regional blood banking. Ph.D. thesis, Northwestern University, Evanston, IL., 1976.

Orda, A., and R. Rom. Shortest-path and minimum-delay algorithm in networks with time-dependent edge-length. *Journal of the Association for Computing Machinery*, 37:603-625, 1990.

Papacostas, C. S., and P. D. Prevedouros. *Transportation engineering and planning*. Prentice-Hall, Inc., New Jersey, 1993.

Papadimitriou, C. H., and K. Steiglitz. *Combinatorial Optimization Algorithms and Complexity*. Prentice-Hall Inc, New Jersey, 1982.

Peeta, S., and H. S. Mahmassani. System optimal and user equilibrium time-dependent traffic assignment in congested networks. *Annals of Operation Research*, 60:81-113, 1995.

Psaraftis, H. N. Dynamic programming approach for sequencing group of identified jobs. *Operations Research*, 28(6):1347-1359, 1980.

Psaraftis H. N. A dynamic programming solution to the single vehicle many-to-many immediate request Dial-A-Ride problem. *Transportation Science*, 14(2):130-154, 1980.

Psaraftis H. N. Analysis of an $O(N^2)$ heuristic for the single vehicle many-to-many Euclidean dial a ride problem. *Transportation Research*, 17B:133-145, 1983a.

Psaraftis H. N. k-Interchange procedures for local search in a precedence-constrained routing problem. *European Journal of Operational Research*, 13:391-402, 1983b.

Rakha, H. A. An evaluation of benefits of user and system optimized route guidance strategies. Master Thesis, Queen's University, Kingston, Ontario, Canada, 1990.

Ran, B., D. E. Boyce, and L. J. LeBlanc. A new class of instantaneous dynamic user-optimal traffic assignment models. *Operations Research*, 41(1):192-202, 1993.

Ran, B., N. M. Rouphail, A. Tarko, and D. E. Boyce. Toward a class of link time functions for dynamic assignment models on signalized networks. *Transportation Research*, 31B:277-290, 1997.

Reinelt, G. The traveling salesman-computational solutions for TSP applications. *Number 840 in lecture notes in computer science*, Springer-Verlag, 1994.

Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. *Object-oriented modeling and design*. Prentice Hall, 1991.

Savelsbergh, M. W. P. Local search routing problems with time-windows. *Annals of Operations Research*, 4:285-305, 1985.

Savelsbergh, M. W. P. A parallel insertion heuristic for vehicle routing with side constraints. *Statistica Neerlandica*, 44(4):139-148, 1990a.

Savelsbergh, M. W. P. An efficient implementation of local search algorithm for constrained routing problems. *European Journal of Operational Research*, 47:75-85, 1990b.

Savelsbergh, M. W. P. The vehicle routing problem with time-windows: Minimizing route duration. *ORSA Journal on Computing*, 4(2):146-154, 1992.

Schildt, H. *The complete reference C++*, 3rd Edition. Osborne McGraw-Hill, Inc., New York, 1998.

Schildt, H. *The complete reference C*, 3rd Edition. Osborne McGraw-Hill, Inc., New York, 1995.

Sexton T., and L. Bodin. Optimizing single vehicle many-to-many operations with desired delivery times: I. Scheduling. *Transportation Science*, 19(4):387-410, 1985a.

Sexton T., and L. Bodin. Optimizing single vehicle many-to-many operations with desired delivery times: II. Routing. *Transportation Science*, 19(4): 411-435, 1985b.

Shaffer, C. A. *A practical introduction to data structures and algorithm analysis*. Prentice-Hall Inc., New Jersey, 1997.

Sheffi, Y. *Urban Transportation Networks*: *Equilibrium Analysis with Mathematical Programming Methods*. Prentice-Hall Inc., New Jersey, 1985.

Sherif, Y. S., and A. K. Erdman. Program solves N-Job, M-Machine sequencing Problem. *Microelectronics and Reliability*, 25(1):55-58, 1985.

Simpson, R. W. An integrated view of air traffic management problems. *Flow control of congested networks*, Edited by A. R. Odoni et al., NATO ASI series, F38:309-322, 1987.

Sherali, H. D., and W. P. Adams. A hierarchy of relaxations between the continuous and convex hull representations for zero-one programming problems. *SIAM J. Discrete Math*., 3:411-430, 1990.

Sherali, H. D., and A. Alameddine. A New Reformulation-Linearization Algorithm for Bilinear Programming Problems. *Journal of Global Optimization*, 2:379-410, 1992.

Sherali, H. D., and C. H. Tuncbilek. A Reformulation-Convexification approach for solving nonconvex quadratic programming problems. *Journal of Global Optimization*, 7:1-31, 1995.

Sherali, H. D., and C. H. Tuncbilek. New reformulation-linearization/convexification relaxations for univariate and multivariate polynomial programming problems. *Operations Research Letters*, 21(1):1-10, 1997.

Sherali, H. D., W. P. Adams, and P. J. Driscoll. Exploiting special structures in constructing a hierarchy of relaxations for 0-1 mixed integer programs. *Operations Research*, 46(3):396-s405, 1998.

Sherali, H. D., and W. P. Adams. *A Reformulation-Linearization Technique for Solving Discrete and Continuous Nonconvex Problems*. Kluwer Academic Publishers, Dordrecht/Boston/London, 1999.

S herali, H. D., and E. L. Brown. A Quadratic Partial Assignment and Packing Model and Algorithm for the Airline Gate Assignment Problem. *Quadratic Assignment and Related Problems, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society,* eds. Pardalos, P. M., and H. Wolkowicz, 16:343-364, 1994.

Sherali, H. D., K. Ozbay, and S. Subramanian. The time-dependent shortest pair of disjoint paths problem: complexity, models, and algorithms. *Networks*, 31:259-272, 1998.

Sherali, H. D., and P. J. Driscoll. On tightening the relaxations of Miller-Tucker-Zemlin formulations for asymmetric traveling salesman problems, manuscript, Department of ISE, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 1997.

Sherali, H. D., and Wang, H. Global Optimization of nonconvex factorable programming problems, manuscript, Department of ISE, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 1998.

Solomon M. M. The minimum spanning tree problem with time-window constraints. *American Journal of Mathematical and Management Sciences*, 6(3):399-421, 1986.

Solomon M. M. Algorithms for the vehicle routing and scheduling problems with time-window constraints. *Operations Research*, 35(2):254-265, 1987.

Solomon M. M., E. K. Baker, and J. R. Schaffer. Vehicle routing and scheduling problems with time-window constraints: efficient implementations of solution improvement procedures. *Vehicle routing*: *Methods and studies,* B. L. Golden and A. A. Assad (Ed.), Elsevier science publishers, North-Holland, 85-105, 1988.

Starfield, A. M., K. A. Amith, and A. L. Bleloch. *How to model it: Problem solving for the computer age*. McGraw-Hill Inc., New York, 1990.

Subramanian, S. *Routing algorithm for dynamic, intelligent transportation network*. Master Thesis, Virginia Polytechnic Institute and State University, 1997.

Syslo, M. M., N. Deo, and J. S. Kowalik. *Discrete Optimization Algorithms with PASCAL programs*. Prentice Hall, New Jersey, 1983.

Taha, H. A. *Integer programming*: *Theory, application and computations*. Academic press, New York, 1975.

Taha, H. A. *Operations research*, 4th edition. McMillan Publishing Company, London, 1987.

Trani, A. A., X. Gu, C. Zhong, and J. Cao. *A system approach to enhance the ground simulation capabilities of SIMMOD*. Research paper, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 1993.

Van Aerde, M. W. *Modeling of Traffic Flows, Assignment and Queueing in Integrated Freeway/Traffic Signal Networks.* Ph.D. Dissertation, University of Waterloo, 1985.

Van Aerde, M. W., and S. Yagar. Dynamic integrated freeway/traffic signal networks: Problems and proposed solutions. *Transportation Research,* 22A:435-444, 1988a.

Van Aerde, M. W., and S. Yagar. Dynamic integrated freeway/traffic signal networks: A routing-based modeling approach. *Transportation Research,* 22A:445-453, 1988b.

Van Aerde, M. W. Class Notes. Unpublished, 1998.

Van Aerde, M. & Associates. INTEGRATION Released for WINDOWS: User's guide Volume I & II: Fundamental model features and advanced model Features". Blackburg, VA, 1999.

Vankatakrishnan, C. S., A. Barnett, and A. R. Odoni. Landings at Logan airport: describing and increasing airport capacity. *Transportation Science*, 27(3):211-227, 1993.

Weijers, T., G. Maters, and P. Aardema. State transition diagram by OMT, Method Engineering Encyclopaedia. Online, Available: http://www.univ-paris1.fr/CRINFO/dmrg/ MEE/misop009/, 1995.

Wie, B. An application of optimal control theory to dynamic user equilibrium traffic assignment. *Transportation Research Record,* 1251:66-73, 1989.

Winston, P. H. *On to C++*. Addison-Wesley, Massachusetts, 1994.

Zhong, Caoyuan. *Modeling of Airport Operations Using an Object-Oriented Approach.* Ph.D. Dissertation, Virginia Polytechnic Institute and State University, 1997.

Ziliaskopoulos, A., and H. S. Mahmassani. Time-Dependent Shortest Path Algorithm for Real-Time Intlligent Vehicle Highway System Applications. *Transportation Research Record*, 1408:94-100, 1993.

Aviation capacity enhancement plan. FAA, Dec. 1998.

Documentation for the Consolidated Operations and Delay Analysis System. Office of aviation policy and plans, FAA, 1997.

Integrated Noise Model (INM) User's guide ver. 5.1. FAA, 1996.

RAMS 2.3 Functional overview. Eurocontrol experimental Centre, 1998.

SIMMOD3 Simulation module: *Software design documentation.* ATAC Corporation, 1993.

SIMMOD3 Simulation module: *Functional specification*. ATAC Corporation, 1993.

Appendix A: List of Input Data

Node data

field 1	serial number
field 2	id number node number for gate: 1-999, node number for taxiway: 1001-9999, node number for runway: 2001-2999.
field 3	node type G(gate), T(Taxiway), R(Runway), I(Intersection).
field 4	x coordinate
field 5	y coordinate

Flight schedule

field 1	serial number
field 2	flight number (ex, AA2456)
field 3	Aircraft type (ex, B727-100)
field 4	operation type D(Departure)/A(Arrival)
field 5	Initial time (hour)
field 6	Initial time (minute)
field 7	Initial time (second)
field 8	Gate number
field 9	Runway number

Edge date

Line 1	field 1	scale
Line 2 -	field 1	serial number
	field 2	from node id
	field 3	to node id
	field 4	Edge type T(Taxiway)/R(Runway)/E(Runway exit)
	field 5	Edge id
	field 6	maximum speed (mph)
	field 7	direction (ex, 2 (ways))

Aircraft data

field 1	Serial number	
field 2	Aircraft type	B727-100
field 3	wheel base (m)	16.23
field 4	OEW (Operating Empty Weight, kg)	72600
field 5	MLW (Maximum Landing Weight, kg)	62400
field 6	CL_{max} (Maximum Lifting Coefficient)	2.59872
field 7	Wing area (m^2)	157.90
field 8	Wing span (m)	32.92
field 9	Length (m)	40.59
field 10	normal taxiing speed (km/hr)	30.0
field 11	maximum taxiing speed (km/hr)	40.0
field 12	maximum acceleration for taxiing (m/s^2)	2.0
field 13	maximum deceleration for taxiing (m/s^2)	-5.0
field 14	//normal takeoff speed (km/hr)	// 300.0
field 15	//maximum acceleration for rolling (m/s^2)	// 9.80
field 16	//maximum acceleration for takeoff (m/s^2)	// 9.80
field 17	velocity at brake-release (ft/s)	27.41
field 18	velocity at lifting-off (ft/s)	230.6
field 19	acceleration at brake-release (ft/s^2)	6.96
field 20	k1	7.053357
field 21	k2 (accel. for takeoff = k1 – k2 *speed)	0.00357250

Appendix B: Constants used in the Simulation Model

```
//Definition.h
#ifndef _DEFINITION_H
#define _DEFINITION_H
//----------------------------------------
// Controls parameters for for Simulation
//----------------------------------------
#define BATCH_JOB      0        // 0: NO, 1: YES  If yes, then use "test.bat".
#define COMMUNICATION  1        // 1: voice channel, 2: data link
#define SEQUENCING     1  // 1: pure FCFS,
                          // 2: FCFS w/ landing priority, or
                          // 3: SWAP squencing
#define spAlgorithm td_SP_2 // static_SP: Static SP
             // td_SP_1  : time-dependent shortest path algorithm for rNode->all, for all timeSlices
             //  td_SP_2     :  time-dependent  shortest  path  algorithm  for  rNode->all,  for  a  SINGLE
timeSlice
// ------------------- CAUTION -------------------
//
// "td_SP_1" may cause a memory deficiency poblem. cause this method takes huge memories such as
//
//                    int   td_shortPath[MAX_NODES][MAX_NODES][NumTimeSlices]
//                    float td_shortTime[MAX_NODES][MAX_NODES][NumTimeSlices]
//
//          Also, takes much more computation time comparing with method 2.
//                    In conclusion, not recommended.

#define PRINT_FLIGHT_STATE      0
#define PRINT_CONTROLLER_STATE  0
// 0: the flight/cotroller state file is not written. (This will save time a lot.)
//-------------------------------
// Definitions for Problem Size
//-------------------------------
#define MAX_NODES     70                // Max. number of Nodes this simulation can handle
#define MAX_FLIGHTS   120               // Max. number of Flights this simulation can handle
#define MaxRunways    10          // Max. number of Runways this simulation can handle
#define MaxNodesPerIntersection 5 // Max. number of Nodes which are connected to one intersection
#define MaxAcftModels 50                // Max. number of AcftModels this simulation can handle
#define InfDistance   99999.9           // Inifite distance for SP algorithm (considered as infinity)
#define MaxNodeIndex  99999     // Max. index for node numbering


//-------------------------------
// Definitions for Simulation Parameters
//-------------------------------
//#define UpdateTDSP    300     // Update TDSP every UpdateTDSP seconds and
                                // NumTimeSlices = SimulationDuration / UpdateTDSP in Simulation

#define dt           (float) 1.0 //second
#define StartHour    7
#define StartMinute  0
#define StartSecond  0
#define EndHour      9
#define EndMinute    0
#define EndSecond    0
#define NumTimeSlices 180        //  Max.  number  of  time  slices  to  keep  the  Time-dependent  link  travel
time for TDSP algorithm
#define  AddLinkTravelTime  3.5      //  (sec)  expected  extra  link  travel  time  for  each  connflicting
flight
#define MaxDelay                600   // (sec.) should be variable to each problem.
#define BufferTimeRatio         1.1   // norminal ROT for departure =
                              // ready time to taxi + normal taxiing time * BufferTimeRatio (=1.1) )
                              // this number is also used in "Random Generation"

//-------------------------------
// UK(ft) -> SI(meter) Unit         // Multiply the conversion factor
//-------------------------------
#define meter_per_ft   0.305    // Conversion factor (ft -> meter)
#define fps_per_mph    1.4667   // Conversion factor (mi/hr -> ft/sec)
#define km_per_mile    1.609    // Conversion factor (mile -> km)
#define kg_per_lb               0.4536   // Conversion factor (pound -> kg)

//-------------------------------
// SI(meter) -> UK(ft) Unit
//-------------------------------
#define ft_per_meter    3.28            // Conversion factor (meter -> ft)
#define mile_per_km   0.62137           // Conversion factor (mile -> km)
#define lb_per_kg         2.2046                  // Conversion factor (kg -> pound)
```

```
	//------------------------------
	// Aircraft Classification Weight
	//------------------------------
	#define WtSmallLarge    60000    // if acft weight < WtSmallLarge (lb) then this is "small"
	#define WtLargeHeavy   300000    // if acft weight > WtLargeHeavy (lb) then this is "heavy"

	//------------------------------
	// Car-Following
	//------------------------------
	#define minGap         15  // minimum gap (second) for coflicting aircraft at Intersection
	                           // if increase this, the TDSP will be more effective.
	#define  SafeDistanceFromCrossToStop  50    //  Safe  Distance  from  the  cross  end  for  aircraft  to  stop.
(ft)
	#define maxLegs 5       // maximum number of legs for a intersection.
	                        //  this  number  is  the  same  to  the  maximum  number  of  conflicting  flights  at  a
intersection
	#define u_f   41.0              //free flow speed (ft/sec) = 45kph
	#define h_j   83.0              //jam headway (ft/veh)
	#define k_h            // design parameter          for headwayControl logic in vehicle-following model
	#define k_s            // design parameter          for speedControl logic in vehicle-following model
	#define reactionTime 1.55       // delta_t                  for micro car-following model
	#define alpha_        .37       // sensitivity paprameter  (for micro car-following model)
	#define normSpeed_mph 18.65 // normal taxiing speed (mph) = 30 (kph)
	#define normSpeed_fps (normSpeed_mph*fps_per_mph) // normal taxiing speed (fps)
	//#define safetyDist

	//------------------------------
	// For Landing Module
	//------------------------------
	//    1) for Flaring Out phase
	#define gamma          (2.75 * 3.1415 / 180)  // radian
	                       // Tangent value of the descent flight path angle (degree, ~N(2.73,0.08^2))
	#define h_th_ft         50.      // Threashold crossing height (ft)  (meter, ~N(15.2,3^2))
	#define g_mps2          9.8    // Acceleration of gravity          (meter/sec^2)
	#define n_fl            1.1       // Flare load factor               (1.1~1.3)
	#define  delta_m              25.0     //  Correction  distance  for  touchdown  diatnce  in  association  with
runway length (m/100 m runway length)
	#define rho             1.1673  // Standard atmosphere air density at 500m        (kg/m^3)
	#define wtFactor        0.5    // weight factor for landing weight (= (OEW + MLW) * wtFactor)

	//    2) for Free Rolling (FR) phase
	#define timeFR         2                  // Time for free rolling (sec)
	#define accFR_mps2     -0.7      // the acceleration for Free Rolling (m/sec^2)

	//    3) for Braking phase
	#define wtForDecSpd                      41000
	#define decisionSpdHeavy_mps    35        // decision speed for heavy aircraft (wt > 41000 lb)
	#define decisionSpdLarge_mps    27        // decision speed for large aircraft (wt < 41000 lb)

	#define accBrakeB727_mps2    -2.19    // decaleration for braking phase (B727) ~N(2.19, 0.416^2)
	#define accBrakeB737_mps2    -2.25    // decaleration for braking phase (B737) ~N(2.25, 0.471^2)
	#define accBrakeB757_mps2    -2.01    // decaleration for braking phase (B757) ~N(2.01, 0.478^2)
	#define accBrakeDC9_mps2     -2.03    // decaleration for braking phase (DC9)  ~N(2.03, 0.414^2)
	#define accBrakeMD80_mps2    -2.05    // decaleration for braking phase (MD80) ~N(2.05, 0.387^2)
	#define accBrakeDefault_mps2 -2.10    // decaleration for braking phase (else) ~N(2.05, 0.387^2)

	//    4) for Coasting phase
	#define EXIT_SPEED_mps           15        // exit speed from the runway
	#define EXIT_SPEED_fps           49.2


	//------------------------------
	//InterEventTimes in communication
	//------------------------------
	#if (COMMUNICATION == 1) // VOICE_CHANNEL

	#define waitngToContactTime      10.0//10.00 // for waiting to the next contact
	                                 // due to either communication jam, or pushback congestion for dep
	#define sendingRequestTime        4.0//11.00       // for sendingRequest       (flight)
	#define receivingRequestTime      4.0//11.00       // for receivingRequest     (controller)
	#define waitingCommandTime        3.0              // for waitingCommand       (flight)
	#define judgingTime               3.0              // for judging              (controller)
	#define receivingCommandTime      4.0//14.00       // for receivingCommand     (flight)
	#define sendingCommandTime        4.0//14.00       // for sendingCommand       (controller)
	#define sendingConfirmationTime   3.0//15.00       // for sendingConfirmation(flight)
	#define recievingConfirmationTime 3.0//15.00       // for receivingConfirmation(controller)

	#else // DATA_LINK

	#define waitngToContactTime     1.0      // for waiting to the next contact
```

```
                                                //  due  to  either  communication  jam,  or  pushback  congestion  for
dep
      #define sendingRequestTime        0.0              // for sendingRequest       (flight)
      #define receivingRequestTime      0.0              // for receivingRequest     (controller)
      #define waitingCommandTime        3.0              // for waitingCommand       (flight)
      #define judgingTime               3.0              // for judging              (controller)
      #define receivingCommandTime      0.0              // for receivingCommand     (flight)
      #define sendingCommandTime        0.0              // for sendingCommand       (controller)
      #define sendingConfirmationTime   0.0              // for sendingConfirmation(flight)
      #define recievingConfirmationTime 0.0              // for receivingConfirmation(controller)

      #endif

      #define   TotalCommunicationTime   (sendingRequestTime   +   waitingCommandTime   +   receivingCommandTime   +
sendingConfirmationTime)
      #define  MaxEarlierContactTime       15                          //  if  flight  contact  no  earlier  than  the
scheduled time
                                                        //   by   MaxEarlierContactTime   then   he   can   continue
to comminicate
      //------------------------------
      //etc.
      //------------------------------
      #define PILOT_SCAN_RATE           3.0              //    Pilot    detect    the    potential    conflict    every
PILOT_SCAN_RATE secconds
      #define TOL_TAXING_SPEED_GAP      0.01             //   Tolerable   speed   gap   between   normalTaxiSpeed   and
currentSpeed
                  // (this is only for the first flight in the link)
                  //   if   abs(normalTaxiSpeed   -   currentSpeed)   >   TOL_TAXING_SPEED_GAP,   then   need   accel   or
decel
      #define maxValue(a, b) (((a) > (b)) ? ((a) : (b)))
      #define minValue(a, b) (((a) > (b)) ? ((b) : (a)))

      #endif
```

Appendix C: Header Files for the Simulation Model

```
//AcftModel.h
#ifndef _ACFTMODEL_H
#define _ACFTMODEL_H
class AcftModel{
public:
    AcftModel();
    AcftModel(char* id, float wb_m, float oew_kg, float mlw_kg,
                        float cl_m, float wa_m2, float ws_m, float l_m,
                        float vT_kph, float vMT_kph, float aMT_mps2, float dMT_mps2,
                        float vTo_kph, float aMR_mps2, float aMTo_mps2,
                        float v1, float v2, float a1, float k1, float k2);
    ~AcftModel();

    //readers
    char* read_id();
    float read_cl_max();
    float read_spdAccelCoeff();

    // SI ------------------------------------------------------------------------------
    float read_wheelBase_m();
    float read_OEW_kg();
    float read_MLW_kg();
    float read_wingArea_m2();
    float read_wingSpan_m();
    float read_length_m();

    //Taxing
    float read_velNormTaxi_kph();         float read_velNormTaxi_mps();
    float read_velMaxTaxi_kph();          float read_velMaxTaxi_mps();
    float read_accMaxTaxi_mps2();
    float read_decMaxTaxi_mps2();
    float read_decNormTaxi_mps2();
    float read_distNormalStop_m();

    // FlaringOut
    float read_timeFlare();

    float read_velApproach_kph();         float read_velApproach_mps();
    float read_velFlare_kph();            float read_velFlare_mps();
    float read_velTouchdown_kph();        float read_velTouchdown_mps();
    float read_accFlare_mps2();

    // FreeRolling
    float read_accFreeRoll_mps2();
    float read_timeFreeRoll();

    // Braking
    float read_decisionSpd_mps();
    float read_accBrake_mps2();

    // Takeoff
    float read_velTakeoff_kph();          float read_velTakeoff_mps();
    float read_accMaxRoll_mps2();
    float read_accMaxTakeoff_mps2();

    // UK ------------------------------------------------------------------------------
    float read_wheelBase_ft();
    float read_OEW_lb();
    float read_MLW_lb();
    float read_wingArea_ft2();
    float read_wingSpan_ft();
    float read_length_ft();

    //Taxing
    float read_velNormTaxi_mph();         float read_velNormTaxi_fps();
    float read_velMaxTaxi_mph();          float read_velMaxTaxi_fps();
    float read_accMaxTaxi_fps2();
    float read_decMaxTaxi_fps2();
    float read_decNormTaxi_fps2();
    float read_distNormalStop_ft();


    // FlaringOut
    float read_velApproach_mph();         float read_velApproach_fps();
    float read_velFlare_mph();            float read_velFlare_fps();
    float read_velTouchdown_mph();        float read_velTouchdown_fps();
    float read_accFlare_fps2();
```

```
        // FreeRolling
        float read_accFreeRoll_fps2();

        // Braking
        float read_decisionSpd_fps();
        float read_accBrake_fps2();

        // Takeoff
        float read_velTakeoff_mph();          float read_velTakeoff_fps();
        float read_accMaxRoll_fps2();
        float read_accMaxTakeoff_fps2();

        float read_v1_fps();         // velocity at brake-release (ft/sec)
        float read_v2_fps();         // velocity at lifting-off (ft/sec)
        float read_a1_fps2();        // acceleration at lifting-off (ft/sec^2)
        float read_k1();             // a = k1 + k2 * v
        float read_k2();

    private:
        char* id_str;
        float spdAccelCoeff,         // spdAccelCoeff(=k)= maxAccel / maxSpeed;
              cl_max;                // max. landing lift coeffcient

        /*-----------------------------------------------
                 SI Unit (The international System of Units)
        -----------------------------------------------*/
        float
                 // Dimensions
                 wheelBase_m,
                 OEW_kg,
                 MLW_kg,
                 wingArea_m2,
                 wingSpan_m,
                 length_m,

                 // Taxing
                 velNormTaxi_kph,  velNormTaxi_mps,             // normal taxing speed
                 velMaxTaxi_kph,   velMaxTaxi_mps,              // maximum taxing speed
                 accMaxTaxi_mps2,                               // maximum accel for taxing
                 decMaxTaxi_mps2,                               // maximum decel for taxing
                 decNormTaxi_mps2,                              // normal  decel for taxing
                 distNormalStop_m,                              // stopping distance at normal taxing speed

                 // Landing
                 // 1) Flaring phase
                 velStall_kph,     velStall_mps,                // stall speed
                 //vCruise_kph,     vCruise_mps,                // normal cruising speed
                 velApproach_kph,  velApproach_mps,             // normal approach speed
                 velFlare_kph,     velFlare_mps,                // threshold crossing speed
                 velTouchdown_kph, velTouchdown_mps,            // normal Touchdown speed
                 distFlare_m,                                   // Flaring distance
                 accFlare_mps2,                                 //       flaring      acceleration      (actually
deceleration)
                 timeFlare,                                     // flaring duration
                 //dFlare_m,                                    // fixed Flare dist, cummulative Flare dist
                 //tFlare,


                 // 2) Free Rolling phase
                 accFreeRoll_mps2,
                 timeFreeRoll,
                 //tFreeRolling,
                 //dFreeRolling_m,

                 // 3) Braking phase
                 decisionSpd_mps,
                 accBrake_mps2,
                 //tBrake,
                 //dBrake_m,

                 // 4) Coasting phase
                 //aCoast_mps2,
                 //dCoast_m,
                 //tCoast_m,

                 // Takeoff
                 velTakeoff_kph,              velTakeoff_mps,    // normal Takeoff speed
                 accMaxRoll_mps2,                                // maximum accel for rolling
                 accMaxTakeoff_mps2;                             // maximum accel for takeoff

        /*-----------------------------------------------
```

```
        UK Unit (The British Units)
    ------------------------------------------------------*/
    float

        // Dimensions
        wheelBase_ft,
        OEW_lb,
        MLW_lb,
        wingArea_ft2,
        wingSpan_ft,
        length_ft,

        // Taxing
        velNormTaxi_mph, velNormTaxi_fps,              // normal taxing speed
        velMaxTaxi_mph,  velMaxTaxi_fps,               // maximum taxing speed
        accMaxTaxi_fps2,                               // maximum accel for taxing
        decMaxTaxi_fps2,                               // maximum decel for taxing
        decNormTaxi_fps2,                              // normal  decel for taxing
        distNormalStop_ft,                            //  stopping  distance  at  normal  taxing  speed


        // Landing

        // 1) Flaring phase
        velStall_mph,     velStall_fps,          // stall speed (constant from equation)
        //vCruise_mph,       vCruise_fps,        // normal cruising speed (constant from equation)
        velApproach_mph,  velApproach_fps,   // normal approach speed (constant from equation)
        velFlare_mph,     velFlare_fps,          //    threshold    crossing    speed    (constant    from
equation)
        velTouchdown_mph,   velTouchdown_fps,      //   normal   Touchdown   speed      (constant   from
equation)
        distFlare_ft,                            // flare dist (constant from equation)
        accFlare_fps2,   // average acceleration during Flaring out phase

        // 2) Free Rolling phase
        accFreeRoll_fps2,

        // 3) Braking phase
        decisionSpd_fps,
        accBrake_fps2,

        // 4) Coasting phase

        //Takeoff
        velTakeoff_mph,     velTakeoff_fps,
        accMaxRoll_fps2,
        accMaxTakeoff_fps2,

        v1_fps,   // velocity at brake-release (ft/sec)
        v2_fps,   // velocity at lifting-off (ft/sec)
        a1_fps2,// acceleration at lifting-off (ft/sec^2)
        k1,       // a = k1 - k2 * v
        k2;
};

#endif
```

```
// Array3D.cpp : Defines the entry point for the console application.
//3DARRAY.h
#ifndef _3DARRAY_H
#define _3DARRAY_H
#include "ostream.h"

template <class type> class TwoDArray{
public:
    TwoDArray(int nRows, int nCols): m_nRows(nRows), m_nCols(nCols){
              m_p2DData = new type[nRows*nCols];
    }
    ~TwoDArray(){
              delete m_p2DData;
    }
    type& operator()(int nRow, int nCol)
    {
              //check dimensions
              if(nRow<0 || nRow>=m_nRows ||         nCol<0 || nCol>=m_nCols)
              {
                        throw "Trying to acces array out of bounds\n";
              }
              return *(m_p2DData + nRow*m_nCols + nCol);
    }
private:
    type*     m_p2DData;
    const int m_nRows;
    const int m_nCols;
};

template <class type> class ThreeDArray{
public:
    ThreeDArray(int nPages, int nRows, int nCols):m_nPages(nPages), m_nRows(nRows), m_nCols(nCols){
              m_p3DData = new type[nPages*nRows*nCols];
              if(m_p3DData == NULL)
                        cout << "Array allocation error: Too big array allocation in ThreeDArray
                                  constructor.\nPlease use the td_SP_2 as a TDSP method\n\n";
    }
    ~ThreeDArray(){
              delete m_p3DData;
    }
    type& operator()(int nPage, int nRow, int nCol)
    {
              //check dimensions
              if(nPage<0 || nPage>=m_nPages ||
                        nRow<0 || nRow>=m_nRows ||
                        nCol<0 || nCol>=m_nCols)
              {
                        cout << "nPage" << nPage << " nRow" << nRow << " nCol" << nCol;
                        throw "Trying to acces array out of bounds\n";
              }
              return *(m_p3DData + nPage*m_nRows*m_nCols + nRow*m_nCols + nCol);
    }
private:
    type*     m_p3DData;
    const int m_nPages;
    const int m_nRows;
    const int m_nCols;
};
#endif
```

```
//Clock.h
#ifndef _CLOCK_H
#define _CLOCK_H

#include "Definition.h"
```

```
class NormalTime{
public:
```

```
    NormalTime();
    ~NormalTime();
    NormalTime(int h, int m, float s);
    void write_NormalTime(int h, int m, float s);
protected:
    int   hour,  minute;
    float second;      // ( float to keep the deci-second )
};


class SystemTime{
public:
    SystemTime();
    ~SystemTime();
protected:
    float systemSecond;                  // ( absolute time starting from 0.0 second)
};

class Clock: private NormalTime, private SystemTime {
public:
    Clock();
    ~Clock();
    Clock(float st);
    Clock(int h, int m, float s);

    // reader and writer for NormalTime
    void  write_Time(float st);
    void  write_Time(int h, int m, float s);
    int        read_hour();
    int   read_minute();
    float read_second();
    float read_systemTime();
    void  convertSystemTimeToNormalTime(float st);
    float convertNormalTimeToSystemTime(int h, int m, float s);

    // reader and writer for SystemTime
    void  advanceTime();
    void  advanceTime(float deltaT);
};

#endif
```

```
//Controller.h
#ifndef _CONTROLLER_H
#define _CONTROLLER_H
#include "AcftModel.h"
#include "ARRAY.h"
#include "Clock.h"
#include "Definition.h"
#include "Edge.h"
#include "Flight.h"
#include "Header.h"
#include "Node.h"
#include "Point.h"
#include "Graph.h"
#include "TaxiEdge.h"
#include "TaxiEdgeList.h"
#include "Type.h"
#include "Clock.h"
#include "Q.h"
#include "Runway.h"



class Controller{
public:

    Controller();
    ~Controller();
    Controller(int totNodes, Flight* f_p, Node* n_p, Graph* FS_p = NULL, Graph* BS_p = NULL);
    virtual void communicate(Flight* f, FlightState_Type currFlightCommState, float sysTime);
    void static_Dijkstra(unsigned int rIndex, int totNodes, ofstream& fStar_out);
    void findStaticPath(Flight* f, int totNodes, unsigned int sPath[]); //find enRoute
    //find & write shortPath
    void findStaticPath(int totNodes, unsigned int rIndex, unsigned int d, unsigned int sPath[]);
    // update the TDSP from this flight's origin to other nodes.
    void td_Dijkstra_Forward(Flight* cFlight_p, int totNodes, ofstream& fowardStar_out);
    void td_Dijkstra_Forward(unsigned int rIndex, int totNodes, ofstream& forwardStar_out);

    // find Time Dependent Shortest Path for all time slices from rNode
    //       -> all nodes, i.e., forward search style
    void td_Dijkstra_Forward_2(float sTime, Flight* cFlight_p, int totNodes, ofstream& fStar_out);
    void    td_Dijkstra_Forward_2(float    sTime,    unsigned    int    rIndex,    int    totNodes,    ofstream&
forwardStar_out);
    //  find  Time  Dependent  Shortest  Path  for  a  single  start  time  slice  from  rNode  ->  all  nodes,
       i.e., forward search style
    int   findTotalConfFlights(Edge*   currEdge_p,   int   tSlice);   //   find   the   number   of   conflicting
                                                                    flights on the all conflicting edges
    void findTDPath(Flight *f, int totNodes, unsigned int td_sPath[], float sTime);
    void  findTDPath(int  totNodes,  unsigned  int  oIndex,  unsigned  int  dIndex,  unsigned  int  td_sPath[],
    int timeSlice);
    void update_NumAcft_TravelTimes(TaxiEdgeList* nTaxiEdgeList_p, float sysTime);
    TaxiEdgeList* make_TaxiEdgeList_p(Flight* f, SP_Algorithm spMethod, int timeSlice = -999);
    TaxiEdgeList*   make_TaxiEdgeList_p(int   oNodeIndex,   int   dNodeIndex,   SP_Algorithm   spMethod,   int
    timeSlice = -999);

    //update the flight strips
    std::list<Flight*> read_pendingFlightList();
    std::list<Flight*> read_processingFlightList();
    std::list<Flight*> read_doneFlightList();

    void pushbackPendingList(Flight* f_p, ofstream& log_out, char* controllerName);
          void     pushbackProcessingList_1(int     sTime,     Flight*     f_p,ofstream&     log_out,     char*
    controllerName1, char* controllerName2);

    // remove from pending, and pushback to processing list
    void pushbackProcessingList_2(int sTime, Flight* f_p, ofstream& log_out, char* controllerName);
    // just pushback to processing list
    void pushbackDoneList(int sTime, Flight* f_p, ofstream& log_out, char* controllerName);
    // remove from processing list, and pushback to done list
    virtual void judge(ofstream& log_out, Flight* f_p, float sysTime);

    //readers
    ControllerState_Type        read_currState();
    float                       read_shortTime(unsigned int i, unsigned int j);
    float                       read_td_shortTimes(unsigned int r, unsigned int d, int tS);
    float                       read_td_shortTime_2(unsigned int r, unsigned int d);
    Flight*                     read_currFlightComm_p();
            Flight*                     read_prevRunwayOccupiedFlight_p();
    float                       read_delayTimeForFlight();
    // writers
    void     write_currState(ControllerState_Type cst);
    void     write_prevRunwayOccupiedFlight_p(Flight* pf);
```

```
        void     write_currFlightComm_p(Flight* cf);
        void     write_delayTimeForFlight(float dTFF);
        void        write_initFlightsList_p(Flight*  f_p);  //  write  the  pointer  to  the  starting  point  of
                                                            Flights linked list
    protected:
        ControllerState_Type currState;
        ControllerMessage_Type decision;
        float delayTimeForFlight;    //if  delayTime  >  0.00,  then  the  flight  will  contact  L/C  again  in
delayTime.

        // for STATIC SHORTEST PATH
        unsigned int shortPath[MAX_NODES][MAX_NODES], //shortest path matrix
                                    sPath[MAX_NODES];   //shortest path from one node to the other nodes
        float    shortTime[MAX_NODES][MAX_NODES]; //travel Time for semi-dynamic
        // for TIME_DEPENDENT SHORTEST PATH
        unsigned int td_sPath[MAX_NODES];
        unsigned int td_shortPath_2[MAX_NODES][MAX_NODES]; //shortest path matrix
        float     td_shortTime_2[MAX_NODES][MAX_NODES]; //travel Time for semi-dynamic
        Clock     nextEventTime;
        Flight* initFlightsList_p,               // pointer to the starting point of Flights linked-list
                  *   prevRunwayOccupiedFlight_p,// previous Flight which occupied runway             (for     the
                                                     min. separation rule)
              *  currFlightComm_p;            // current communicating Flight
        Node*       nodes_p;                  // pointer to the starting point of Nodes array
        Edge**      initEdgeNodes_p;          // pointer to the starting point of EdgeNodes array
        Graph*      forwardStar_p,            // pointer to the airport network with forwardStar_p
             *      backwardStar_p;           // pointer to the airport network with backwardStar_p
        std::list<Flight*> pendingFlightList,
                  processingFlightList,
                  doneFlightList;
        std::list<Flight*>::iterator p;
    };



    /*************** GROUND CONTROLLER *********************/
    class GroundController: public Controller{
    public:
        GroundController();
        GroundController(int totNodes, Flight* f_p, Node* n_p, Graph* FS_p = NULL, Graph* BS_p = NULL);
        ~GroundController();
        virtual   void   communicate(ofstream&   log_out,   Flight*   f_p,   FlightState_Type   currFlightCommState,
        float sysTime);
        virtual void judge(ofstream& log_out, Flight* f_p, float sysTime);
    private:
        TaxiEdgeList* newTaxiEdgeList_p;               // pointer to the starting point of New TaxiEdge list
    };

    /*************** LOCAL CONTROLLER *********************/
    class LocalController: public Controller{
    public:
        LocalController();
        LocalController(int  totNodes,  Flight*  f_p,  Node*  n_p,  int  totRwy,  Runway  **rwy_p,  Graph*  FS_p  =
        NULL, Graph* BS_p = NULL);
        ~LocalController();
              virtual     void     communicate(ofstream&      log_out,      Flight*      f_p,      FlightState_Type
        currFlightCommState, float sysTime);
        virtual void judge(ofstream& log_out, Flight* f_p, float sysTime);
        float    judgeForClearance(float    sysTime);//decide    the    clearance    for    takeoff/landing    with
                                                    delayTimeForFlight if needed.
        void write_prevFlight_p(Flight* pF);
        void write_currFlight_p(Flight* cF);
        void write_nextFlight_p(Flight* nF);
        Flight* read_prevFlight_p();
        Flight* read_currFlight_p();
        Flight* read_nextFlight_p();
    private:
        int totRunway;
        Runway **runway_p;
        TaxiEdgeList* newRunwayEdgeList_p;        //  pointer  to  the  starting  point  of  New  RunwayEdge  list
                                                  for Takeoff/Landing
        Flight *prevFlight_p,   //previous flight on the runway operation
                    *currFlight_p,   //currnet flight on the runway operation
                    *nextFlight_p;   //next flight on the runway operation
        float minSep[4][3][3];  // Min. Separtion Rule
    };

    #endif
```

```
//Edge.h

#ifndef _EDGE_H
#define _EDGE_H
#include "Type.h"
#include "Definition.h"
#include "Node.h"
#include <Math.h>  //for the "sqrt" and "pow"

class Flight;
class Edge{
public:
    Edge();
    ~Edge();
    Edge(float scale, char* id_str, char type, Node* f, Node* t, float r);
    // readers
    char*     read_id_str_p();
    Edge_Type read_edgeType();
    Node*     read_fromNode_p();
    Node*     read_toNode_p();
    int       read_fromNodeId();
    int       read_toNodeId();
    int       read_fromNodeIndex();
    int       read_toNodeIndex();
    float     read_restrict();
    float     read_distance_ft();
    float     read_distance_m();
    float     read_travelTime();
    float*    read_travelTimes();                  // return the pointer to array of "the travelTimes"
    float     read_travelTimes(int tSlice);
    Flight*   read_flightInEdge_p();      // return the pointer to the flight in the edge
              int         read_numFlightInEdge();
    int         read_numFlightInEdge(int tSlice); //
    Edge*     read_nextEdge_p();
    Edge*     read_nextBSEdge_p();
    // writers
    void       write_distance_ft(float d_ft);
    void       write_distance_m(float d_m);
    void      write_travelTime(float tt);
    void      add_numAcft(int tSlice);
    void      write_travelTimes(int tSlice, float tt);
    void      write_nextEdge_p(Edge* nxt_p);
    void      write_nextBSEdge_p(Edge* nxtBS_p);
    void      write_flightInEdge_p(Flight* tempFlightInEdge_p);
    void      write_numFlightInEdge(int nFIE);
private:
    char       *id_str_p;
    Edge_Type  edgeType;
    float      distance_ft,
               distance_m,
               travelTime,
               restrict;        //e.g. exit speed if this edge is exit edge.
    float     *travelTimes;     // For Time dependent SP algorithm (Variable Array)
    //CurrentEdgeDirection_Type currentDirection;
    Node       *fromNode_p,
              *toNode_p;
    Flight    *flightInEdge_p;
    int       numFlightInEdge,
               *numAcft;         //  #  of  vehicles  in  the  edge    for  Time  dependent  SP  algorithm
(Variable Array)
    Edge      *nextEdge_p,       //  Pointer to NextEdge in forwardStar
               *nextBSEdge_p;    //  Pointer to NextEdge in backwardStar
};

#endif
```

```
//Flight.h
#ifndef _FLIGHT_H
#define _FLIGHT_H
#include "Type.h"
#include "Point.h"
#include "AcftModel.h"
#include "Header.h"
#include "Point.h"
#include "Position.h"
#include "Clock.h"
#include "Graph.h"
#include "Edge.h"
#include "Runway.h"
#include "TaxiEdge.h"
#include "TaxiEdgeList.h"
#include "Definition.h"

class GroundController;
class LocalController;
class Flight{
public:
    Flight(char* fId_p, AcftModel* mdl_p, char oper,      int sHour, int sMin, float sSec,
       int hour_norm, int min_norm, float sec_norm,   int hour_fcfs, int min_fcfs, float sec_fcfs,
       int hour_swap, int min_swap, float sec_swap,   int gIndex, char *rId_p,
       Runway** runway_p, int totRunway, Node* nodes, Graph* FS_p, Graph* BS_p, Flight* pFIL_p,
       GroundController* GC_p, LocalController* LC_p);
    ~Flight();

    //Communication
    bool checkNeedToComm(float nxtCommEventTime, float sysTime);
    void communicateGroundController(ofstream& log_out, float sysTime);
    void communicateLocalController(ofstream& log_out, float sysTime);

    //TakeOff
    void roll();
    void liftOff();

    //Landing
    void flareOut();
    void freeRoll();
    void brake();
    void coast(float dToExit);
    void exitRunway(float dToExitEnd);
    void parkingForArrival();
    Node* findExitNode_p();
    float findDistToExit();
    float findDistToTaxiDestination();
    float findDistToTheCross();

    //Taxing
    void taxiToDest(Clock sysTime, float scale, ofstream& log_out);
    void move(TaxiEdgeList* crrEdgeList_p, int desNodeIndex);
    void deQueue(Edge* currTaxiEdge_p);
    void enQueue(Edge* nextTaxiEdge_p);
    float computeNextAccel_1();
    float computeNextAccel_2(float d, float v_fnl);
    float computeNextAccel_3(Clock sysTime);
    float computeSpdAfterD(float distToNextExit);
    float computeNextTaxiAccel(Clock sysTime, float scale, ofstream& log_out);
    void  collisionDetectResolve(Clock sysTime, float scale, ofstream& log_out);
    float carFollowing();

    //Statistics
    void statistics(ofstream& log_out);

    // For sorting the STL list (see 651 pp., in "complete reference C++")
    // The required operator overloadings are dependent on the compiler.
    friend bool operator <  (Flight &o1, Flight &o2);
    friend bool operator >  (Flight &o1, Flight &o2);
    friend bool operator == (Flight &o1, Flight &o2);
    friend bool operator != (Flight &o1, Flight &o2);

    //Readers
    char*              read_flightId_p();
    char*              read_runwayId_p();
    AcftModel*         read_acftModel_p();
    Operation_Type     read_operationType();
    int                read_orgNodeIndex();
    int                read_desNodeIndex();
    int                read_gateNodeIndex();
    int                read_exitEnterNodeIndex();
```

```
int                 read_currNodeIndex();
int                 read_nextNodeIndex();
Point               read_currPoint();
float               read_currSpeed();
Position*           read_currPosition_p();
float               read_nextAccel();
ControllerMessage_Type       read_permission();
Priority_Type                read_priority();
ConflictChecked_Type         read_conflictChecked();
Flight*             read_conflictingFlight_p();
float               read_depTimeAtCross();
float               read_nextCommEventTime();
float               read_nextMoveEventTime();
float               read_rotTimeRecommended();
float               read_pushbackTimeRecommended();
float            read_rotTimeNorm();
float               read_rotTimeFcfs();
float               read_rotTimeSwap();
float               read_phaseTimeStamp_b();
float               read_phaseTimeStamp_e();
float               read_linkTimeStamp_b();
float               read_linkTimeStamp_e();
float               read_taxingTimeStamp_b();    // the begin/end time stamp of taxing
float               read_taxingTimeStamp_e();
float              read_rotTimeStamp_b();
float              read_rotTimeStamp_e();
float               read_GC_TimeStamp_b();
float               read_GC_TimeStamp_e();
float               read_LC_TimeStamp_b();
float               read_LC_TimeStamp_e();
float               read_taxingDelay_sec();      // delay due to taxiway congestion
         float              read_runwayDelay_sec();      // delay due to runway congestion
         float              read_commGCDelay_sec();      // delay due to communication jam
         float              read_commLCDelay_sec();      // delay due to communication jam
float               read_travelTime();
float               read_lastTaxiEdgeLength_ft();
GroundController*   read_groundController_p();
LocalController*    read_localController_p();
Graph*              read_forwardStar_p();
Graph*              read_backwardStar_p();
TaxiEdgeList*       read_taxiEdgeList_p();
TaxiEdgeList*       read_runwayEdgeList_p();
Edge*               read_currTaxiEdge_p();
Edge*               read_currRunwayEdge_p();
Node*               read_exitNode_p();
Flight*             read_leadingFlightInEdge_p();
Flight*             read_followingFlightInEdge_p();
Flight*             read_nextFlightInList_p();
Flight*             read_prevFlightInList_p();
FlightState_Type    read_currMoveState();
FlightState_Type    read_currCommState();
int                 read_numTrialToContactLC();

int                 read_numTrialToContactGC();


//Writers
friend ofstream &operator<<(ofstream &stream, Flight f);
void write_currMoveState(FlightState_Type fst);
void write_currCommState(FlightState_Type fst);
void write_currSpeed(float cSpd);
void write_permission(ControllerMessage_Type pt);
void write_priority(Priority_Type pr);
void write_conflictChecked(ConflictChecked_Type cC);
void write_conflictingFlight_p(Flight* cF);
void write_depTimeAtCross(float dTAC);
void write_nextAccel(float nxtAccel);
void write_systemInputTime(float sysInputTime);
void write_nextCommEventTime(float nextCETime);
void write_nextMoveEventTime(float nextMETime);
void write_startTime(float stTime);                      // write the simulation input time.
void write_phaseTimeStamp_b(float sysTime);
void write_phaseTimeStamp_e(float sysTime);
void write_linkTimeStamp_b(float sysTime);
void write_linkTimeStamp_e(float sysTime);
void write_taxingTimeStamp_b(float sysTime);             // the begin/end time stamp of taxing
void write_taxingTimeStamp_e(float sysTime);
void write_rotTimeStamp_b(float sysTime);
void write_rotTimeStamp_e(float sysTime);
void write_GC_TimeStamp_b(float sysTime);
void write_GC_TimeStamp_e(float sysTime);
```

```
        void write_LC_TimeStamp_b(float sysTime);
        void write_LC_TimeStamp_e(float sysTime);
        void write_taxingDelay_sec(float tTD);
        void add_numTrialToContactLC();                  // add by 1
        void subtract_numTrialToContactLC();             // subtract by 1
        void add_numTrialToContactGC();                  // add by 1
        void subtract_numTrialToContactGC();             // subtract by 1
        void add_taxingDelay_sec(float tD_sec); // delay due to taxiway congestion
        void  add_runwayDelay_sec(float  rD_sec);  //  delay  due  to  runway  congestion  (not  used  for  the  time
being)
        void add_commGCDelay_sec(float cGCD_sec);         // delay due to communication jam
             void add_commLCDelay_sec(float cLCD_sec);   // delay due to communication jam
        void write_lastTaxiEdgeLength_ft(float lTEL_ft);
        void write_taxiEdgeList_p(TaxiEdgeList *newEdgeList_p);
        void write_runwayEdgeList_p(TaxiEdgeList *newEdgeList_p);
        void write_currTaxiEdge_p(Edge*       cTaxiEdge_p);
        void write_currRunwayEdge_p(Edge* cRunwayEdge_p);
        void write_orgNodeIndex(int oNI);
        void write_desNodeIndex(int dNI);
        void write_exitNode_p(Node* eNode_p);
        void write_currPosition_p(Position* cPosition_p);
        void write_groundController_p(GroundController* gc_p);
        void write_localController_p (LocalController* lc_p);
        void write_leadingFlightInEdge_p(Flight* lFIE_p);
        void write_nextFlightInList(Flight* nxt_p);
        void write_prevFlightInList(Flight* prv_p);
        void write_rotTimeRecommended(float rotRec);
        void write_pushbackTimeRecommended(float pbRec);

    protected:
        Flight(); // constructor. never be used.
    private:
        char*              flightId_p,
           *               runwayId_p;
        AcftModel*         acftModel_p;
        Operation_Type     operationType;
        FlightState_Type   currMoveState,
                           currCommState;
        bool             everStoppedInLastEdge;
        ControllerMessage_Type permission;
        Priority_Type            priority;
        ConflictChecked_Type   conflictChecked;
        Flight*                conflictingFlight_p;// Conflicting flight for taxing
        Clock                    depTimeAtCross;
        int                orgNodeIndex,
                           desNodeIndex,
                           currNodeIndex;
        Node*            gateNode_p;
        Position*          currPosition_p;
        float              currSpeed,
                           nextAccel;
        Clock              systemInputTime,
                           nextCommEventTime,
                           nextMoveEventTime,
                           pushbackTimeRecommended,     // for deprtures
                           rotTimeRecommended,          // assigned to one of rotTimeNorm or rotTimeSwap.
                           rotTimeNorm,
                           rotTimeFcfs,
                           rotTimeSwap,
                           linkTimeStamp_b,     linkTimeStamp_e,
                           phaseTimeStamp_b,    phaseTimeStamp_e,
                           taxingTimeStamp_b, taxingTimeStamp_e,
                           GC_TimeStamp_b,      GC_TimeStamp_e,
                           LC_TimeStamp_b,      LC_TimeStamp_e,
                           rotTimeStamp_b,      rotTimeStamp_e;
        int                        numTrialToContactGC,             //  number  of  trials  to  contact  ground
                controller
                           numTrialToContactLC;      // number of trials to contact local  controller
        float              taxingDelay_sec,             // delay due to taxiway congestion
                           runwayDelay_sec,             // delay due to runway congestion
                           commGCDelay_sec,             //   delay   due   to   communication   jam   with   Ground
Controller
                           commLCDelay_sec;             //   delay   due   to   communication   jam   with   Local
Controller
        float              travelTime,
                           lastTaxiEdgeLength_ft;
        GroundController*  groundController_p;
        Graph             *forwardStar_p,
                          *backwardStar_p;              // for conflict resoltion and (un)blocking the link
        TaxiEdgeList     *taxiEdgeList_p;               // for Taxing path
        Edge               *currTaxiEdge_p;             // The curr Edge in Taxi path.
```

```
    LocalController  *localController_p;
    TaxiEdgeList       *runwayEdgeList_p;          // for Takeoff/Landing path
    Edge               *currRunwayEdge_p;          // The curr Edge in Runway path.
    Node             *exitNode_p;                  // exit Node


    Flight *leadingFlightInEdge_p,                 // Leading Flight in a CuurEdge
       *followingFlightInEdge_p,                   // Following Flight in a CuurEdge
       *prevFlightInList_p,                        // prev Flight in Flight List
       *nextFlightInList_p;                        // Next Flight in Flight List

};
#endif



//Graph.h

#ifndef _GRAPH_H
#define _GRAPH_H

#include "Type.h"
#include "Definition.h"
#include "Node.h"
#include "Node.h"
#include "Edge.h"

typedef Edge* Edge_p;
class Graph{
public:
    Graph(int totNodes, int totEdges);
    ~Graph();
    int n();
    int e();
    Edge* first(int nodeIndex); // get the first edge for a node
    bool isEdge(Edge* edg_p);    // return 1 if (edge != null)
    Edge* next(Edge* edg_p);     // get the next edge for a node
    Edge* nextBS(Edge* edgBS_p);
    int fromNodeId(Edge* edg_p);// get the fromNodeId for this edge
    int toNodeId(Edge* edg_p);  // get the toNodeId for this edge
    Edge*  Graph::findEdge_p  (int  fnIndex,  int  tnIndex);  //  find  the  pointer  to  the  edge  from  fnIndex
                                                                to tnIndex
    float  tTime (int fnIndex, int tnIndex);       //  find  the  travel  time  for  the  edge  from fnIndex to
tnIndex
    float*  tTimes(int  fnIndex,  int  tnIndex);   //  find  the  pointer  to  the  travel  time  matrix  for  the
                                                        edge from fnIndex to tnIndex
    float  Dist  (int fnIndex, int tnIndex);       //  find  the  distance  for  the  edge  from  fnIndex  to
tnIndex
    float  tTime(Edge* edg_p);                     // find the travel time for edge
    float* tTimes(Edge* edg_p);                    // find the travel time matrix for edge
    float  Dist (Edge* edg_p);                     // find the diatnce for edge
    //readers
    Edge*  read_list(int fnIndex);  //return the pointer to (Edge*) list[fnIndex]
    Edge** read_list();                                  //return the pointer to (Edge**)list
    //writers
    void write_list(int fnIndex, Edge* newEdge_p); // list[fnIndex] = newEdge_p;
    void write_totNodes(int tN);                                        // write total Nodes
    void write_totEdges(int tE);                  // write total Edges
private:
    Edge_p* list;
    int totNodes;
    int totEdges;
    VisitMark_Type* visitMark;
};

#endif _GRAPH_H
```

```
// Header.h
#ifndef _HEADER_H
#define _HEADER_H
#include<conio.h>
#include<stdio.h>
#include<fstream.h>
#include<iostream.h>
#include<istream.h>
#include<ostream.h>
#include<stdlib.h>
#include<string.h>
//#include<search.h>
#include<math.h>
#include<assert.h>
#include<iomanip.h>

#include <list>
#include <algorithm>

#endif



//Node.h
#ifndef _NODE_H
#define _NODE_H
#include "Type.h"
#include "Header.h"
#include "Point.h"

class Flight;
class Edge;

class Node{
public:
    Node();
    ~Node();
    Node(int ind, int id, char type, Point point, float p=0.0);
    Node(int ind, int id, char type, float x, float y, float p=0.0);
    Node(int ind, int id, char type, float x, float y, float p, Flight* fin_p);
    //readers
    int        read_index();
    int        read_id();
    float      read_x();
    float      read_y();
    Node_  Type  read_nodeType();
    float      read_passingTime();
    Point&  read_pt();
              Bool     read_haveEverEntered(); // {true, false}; for DEQueue in TDSP
    Bool     read_isInQueue();       // {true, false}; for DEQueue in TDSP
    //writers
    void      write_fs_p(Edge* f_p);
    void      write_bs_p(Edge* b_p);
    void      write_haveEverEntered(bool hEE);
    void      write_isInQueue(bool iIQ);
private:
    int index,                    // serial index (= 1,2,3,...)
        id;                       // id (= User defined id) ex, 2001, 2005,...
    Node_Type nodeType;
    bool haveEverEntered,         // {true, false}; for DEQueue in TDSP
         isInQueue;
    float passingTime;
    Point pt;
    Flight* flightInNode_p;
    Edge* fs_p,                //pointer to ForewardStar
        * bs_p;                   //pointer to BackwardStar
};
#endif
```

```
//Point.h
#ifndef _POINT_H
#define _POINT_H
class Point{
public:
    Point();
    ~Point();
    Point(float x1, float y1);
    float read_x();
    float read_y();
    void write_x(float x1);
    void write_y(float y1);
private:
   float x, y;
};
#endif


//Position.h
#ifndef _POSITION_H
#define _POSITION_H
#include "Point.h"
class Position{
public:
    Position();
    ~Position();
    Position(float x1, float y1, float tDT = 0.0);
    Position(Point& p, float tDT = 0.0);
    //readers
    float read_traveledDistInLink();
    float read_traveledDistTotal();
    float read_currX();
    float read_currY();
    //writers
    void write_traveledDistInLink(float tDistL);
    void write_traveledDistTotal(float tDistT);
    void write_currX(float);
    void write_currY(float );
private:
    float traveledDistInLink,
                traveledDistTotal;
    Point pt;
};
#endif _POSITION_H
```

```
//Q.h  (for the SP algorithm)
#ifndef _Q_H
#define _Q_H
#include "Node.h"
#include "definition.h"
class NodeList{  // Doubly-linked lists (100-shaffer)
public:
    int      nodeIndex;
    NodeList* prevNodeList_p,
              * nextNodeList_p;
    NodeList(int nIndex, NodeList* prv_p = NULL, NodeList* nxt_p = NULL);
    NodeList(NodeList* prv_p = NULL, NodeList* nxt_p = NULL);
    ~NodeList();
};
class Q{      //115page in Shaffer's text book
public:
    Q();
    ~Q();
    int  firstNodeIndex();        // return the first node's index
    int  deQueueFirst();       // return the from-node's index of the first Node in the queue.
    int    deQueueMin(int  r,  float  shortTime[][MAX_NODES]);  //  find  the  minimum  distance  node  from
                              root  node,  r,  to  to-nodes  in  the  (Node)  queue  and  return  the  to-node's
                              index
    void enQueueFirst(int nIndex);
    void enQueueLast(int nIndex);
    void deleteNodeList(NodeList* nl_p);
    bool isEmpty() const;
    int  totCurrElements() const;
    void clear();
private:
    NodeList* headNodeList_p,
              * tailNodeList_p,
              * currNodeList_p;
    void deleteNode(int* n_p);
};
#endif



////Runway.cpp

#ifndef _RUNWAY_H
#define _RUNWAY_H
#include "TaxiEdge.h"
#include "Header.h"
#include "Definition.h"
class Runway{
public:
    Runway();
    Runway(char* rId_p, TaxiEdge* rLink_p);
    ~Runway();
    char*       read_runwayId();
    TaxiEdge* read_runwayLink_p();
    TaxiEdge* read_nextRunwayLink();
private:
    char* id_p;
    TaxiEdge* runwayLink_p;  // pointer to starting link
                        // next runwayLink = runwayLink_p->read_nextTaxiEdge_p();
};
#endif
```

```
// Simulation.h
#ifndef _SIMULATION_H
#define _SIMULATION_H
#include "Header.h"
#include "Type.h"
#include "Definition.h"
#include "Clock.h"
#include "Point.h"
#include "Node.h"
#include "Edge.h"
#include "Graph.h"
#include "Runway.h"
#include "AcftModel.h"
#include "Controller.h"
#include "Flight.h"
#include "Q.h"
#include "ARRAY.h"

#include<ostream.h>

        ThreeDArray<unsigned       int>       td_shortPath(MAX_NODES,       MAX_NODES,       NumTimeSlices);
           //travelPath by keeping the next node to destination
        ThreeDArray<float>      td_shortTime(MAX_NODES, MAX_NODES, NumTimeSlices);      //travelTime

        float totTaxingTime_sec  = 0.0;
        float totTaxingDelay_sec = 0.0;
        float totRunwayDelay_sec = 0.0;
        float totCommLCDealy_sec = 0.0;
        float totCommGCDealy_sec = 0.0;

        typedef Edge* Edge_p;
        int comp(const void *a , const void *b);
        struct FlightData{
        int       sn;                   // serial number
        char      *id_str,     // AA234
              *model_str;               // B757-300
        AcftModel *acftModel_p;
        char      operationType;        // char ('A' or 'D')
        int       startHour,   // 7  (o'clock)
             startMin;         // 6  (minute)
        float     startSec;       // 34.0 (second)
        int  rotHour_norm,     // 7  (o'clock)
             rotMin_norm;      // 6  (minute)
        float     rotSec_norm;          // 34.0 (second)
        int  rotHour_fcfs,     // 7  (o'clock)
             rotMin_fcfs;      // 6  (minute)
        float         rotSec_fcfs;      // 34.0 (second)
        int  rotHour_swap,     // 7  (o'clock)
             rotMin_swap;      // 6  (minute)
        float         rotSec_swap;      // 34.0 (second)
        float     startTime;
        int       gateId,         // Gate Node Number = 7
                  gateIndex;
        char*     runway_str;           // Runway, ex, R15
    };

    ofstream& operator<<(ofstream &stream, Flight f){
              if(f.taxiEdgeList_p != NULL || f.runwayEdgeList_p != NULL){
                        stream //<< f.sysTime.read_systemTime()
                                  << setw(10)
                                  << setprecision(7)
                        << f.read_flightId_p() << "  "
                        << setiosflags(ios::showpoint) << "("
                        << f.currPosition_p->read_currX() << ", "
                        << f.currPosition_p->read_currY() << ")  ";
              switch (f.currCommState){
                        case readyToCommunicate:     stream << " readyToCommunicate    ";  break;
                        case waitNextContact:                stream << " waitNextContact       ";  break;
                        case sendingRequest:                 stream << " sendingRequest        ";  break;
                        case waitingCommand:                 stream << " waitingCommand        ";  break;
                        case receivingCommand:               stream << " receivingCommand      ";  break;
                        case sendingConfirmation:    stream << " sendingConfirmation   ";  break;
                        case waitControllerContact:          stream << " waitControllerContact ";  break;
                        default:                     stream << " somthing wrong in currCommState ";
              }
              switch (f.permission){
                        case unDecided:              stream << " unDecided        ";  break;
                                                             stream << " fileApproved     ";  break;
                        case reRoute:                stream << " reRoute          ";  break;
                        case stopThere:              stream << " stopThere        ";  break;
                        case waitThere:              stream << " waitThere        ";  break;
```

```
                case reStart:                       stream << " reStart         ";        break;
                case clearToPushback:               stream << " clearToPushback  ";  break;
                case clearToTaxi:           stream << " clearToTaxi      ";        break;
                case clearToTakeOff:                stream << " clearToTakeOff   ";        break;
                case clearToLand:           stream << " clearToLand      ";        break;
                default:                    stream << " somthing wrong in currPermission ";
        }

        switch (f.currMoveState){
                case parking:               stream << " parking "       ;  break;
                case pushingBack:           stream << " pushingBack    ";  break;
                case taxingToDepartureQueue:          stream << " taxingToDepQue ";  break;
                case waitingToTaxi:                   stream << " waitingToTaxi  ";  break;
                case delayedAtCross:                  stream << " delayedAtCross ";  break;
                case speedUp:               stream << " speedUp        ";  break;
                case steadyTaxing:                    stream << " steadyTaxing   ";  break;
                case slowDown:              stream << " slowDown       ";  break;
                case turning:                         stream << " turning        ";  break;
                case waitingOnRunway:                 stream << " waitingOnRnwy  ";  break;
                case rolling:                         stream << " rolling        ";  break;
                case liftingOff:            stream << " liftingOff     ";  break;
                case endOfDeparture:                  stream << " endOfDeparture ";  break;

                case onFinal:               stream << " onFinal        ";  break;
                case flaringOut:            stream << " flaringOut     ";  break;
                case freeRolling:           stream << " freeRolling    ";  break;
                case braking:               stream << " braking        ";  break;
                case coasting:              stream << " coasting       ";  break;
                case exitingRunway:                   stream << " exitingRnwy    ";  break;
                case taxingToGate:                    stream << " taxingToGate   ";  break;

                default:                    stream << " somthing wrong in currMoveState ";
        }

        stream
                << f.currSpeed << "  "
                << f.nextAccel << "  "
                << setw(5);

        if(f.taxiEdgeList_p != NULL){
                stream
                << f.taxiEdgeList_p->currEdge_p()->read_fromNode_p()->read_id() << " -> "
                << f.taxiEdgeList_p->currEdge_p()->read_toNode_p()->read_id()   << "  "
                << setw(10)
                << f.taxiEdgeList_p->currEdge_p()->read_distance_ft()    << "  ";
        }
        else if(f.runwayEdgeList_p != NULL){
                stream
                << f.runwayEdgeList_p->currEdge_p()->read_fromNode_p()->read_id() << " -> "
                << f.runwayEdgeList_p->currEdge_p()->read_toNode_p()->read_id()   << "  "
                << setw(10)
                << f.runwayEdgeList_p->currEdge_p()->read_distance_ft()    << "  ";
        }
        else stream << " somthing wrong in node -> node ";


        stream
                << f.currPosition_p->read_traveledDistInLink() << "  "
                << f.currPosition_p->read_traveledDistTotal()  << "  "
                << endl;
}// if
else{
        stream //<< f.sysTime.read_systemTime()
                << setw(10)
                << setprecision(7)
       //<< setiosflags(ios::fixed)
                << f.read_flightId_p() << "  "
                << setiosflags(ios::showpoint) << "("
                << f.currPosition_p->read_currX() << ", "
                << f.currPosition_p->read_currY() << ") ";

        switch (f.currCommState){
                case readyToCommunicate:    stream << " readyToCommunicate    ";  break;
                case waitNextContact:               stream << " waitNextContact       ";  break;
                case sendingRequest:                stream << " sendingRequest        ";  break;
                case waitingCommand:                stream << " waitingCommand        ";  break;
                case receivingCommand:              stream << " receivingCommand      ";  break;
                case sendingConfirmation:   stream << " sendingConfirmation   ";  break;
                case waitControllerContact:         stream << " waitControllerContact ";  break;
                default:                    stream << " somthing wrong in currCommState ";
        }
```

```
            switch (f.permission){
                     case unDecided:             stream << " unDecided        ";  break;
                     case fileApproved:                  stream << " fileApproved     ";  break;
                     case reRoute:               stream << " reRoute          ";  break;
                     case stopThere:             stream << " stopThere        ";  break;
                     case waitThere:             stream << " waitThere        ";  break;
                     case reStart:                       stream << " reStart          ";        break;
                     case clearToPushback:               stream << " clearToPushback  ";  break;
                     case clearToTaxi:           stream << " clearToTaxi      ";        break;
                     case clearToTakeOff:                stream << " clearToTakeOff   ";        break;
                     case clearToLand:           stream << " clearToLand      ";        break;
                     default:                    stream << " somthing wrong in currPermission ";
            }

            switch (f.currMoveState){
                     case parking:               stream << " parking        ";  break;
                     case pushingBack:           stream << " pushingBack    ";  break;
                     case taxingToDepartureQueue:          stream << " taxingToDepQue ";  break;
                     case waitingToTaxi:                 stream << " waitingToTaxi  ";  break;
                     case speedUp:               stream << " speedUp        ";  break;
                     case steadyTaxing:                  stream << " steadyTaxing   ";  break;
                     case slowDown:              stream << " slowDown       ";  break;
                     case turning:                       stream << " turning        ";  break;
                     case waitingOnRunway:               stream << " waitingOnRnwy  ";  break;
                     case rolling:                       stream << " rolling        ";  break;
                     case liftingOff:            stream << " liftingOff     ";  break;
                     case endOfDeparture:                stream << " endOfDeparture ";  break;

                     case onFinal:               stream << " onFinal        ";  break;
                     case flaringOut:            stream << " flaringOut     ";  break;
                     case freeRolling:           stream << " freeRolling    ";  break;
                     case braking:               stream << " braking        ";  break;
                     case coasting:              stream << " coasting       ";  break;
                     case exitingRunway:                 stream << " exitingRunway  ";  break;
                     case taxingToGate:                  stream << " taxingToGate   ";  break;

                     default:                    stream << " somthing wrong currMoveState ";
            }
            stream
                     << f.currSpeed << "  "
                     << f.nextAccel << "  "

                     << setw(5)
                     << 0 << " -> "
                     << 0 << "  "
                     << setw(10)
                     << 0 << "  "

                     << f.currPosition_p->read_traveledDistInLink() << "  "
                     << f.currPosition_p->read_traveledDistTotal()  << "  "
                     << endl;
    }// else
    return stream;
};

bool operator<(Flight &o1, Flight &o2)
  { return o1.rotTimeRecommended.read_systemTime() <  o2.rotTimeRecommended.read_systemTime(); }
bool operator>(Flight &o1, Flight &o2)
  { return o1.rotTimeRecommended.read_systemTime() >  o2.rotTimeRecommended.read_systemTime(); }
bool operator==(Flight &o1, Flight &o2)
 { return o1.rotTimeRecommended.read_systemTime() == o2.rotTimeRecommended.read_systemTime(); }
bool operator!=(Flight &o1, Flight &o2)
 { return o1.rotTimeRecommended.read_systemTime() != o2.rotTimeRecommended.read_systemTime(); }

#endif
```

```
// TaxiEdge.h

#ifndef _TAXIEDGE_H
#define _TAXIEDGE_H
#include "Edge.h"

class TaxiEdge{
public:
    TaxiEdge();
    TaxiEdge(Edge* const newEdge_p, TaxiEdge* nxt = NULL);
    ~TaxiEdge();
    Edge*              read_edgeEle_p();
    TaxiEdge*          read_nextTaxiEdge_p();
    void write_nextTaxiEdge_p(TaxiEdge* tempNextTaxiEdge_p);
private:
    Edge*              edgeEle_p;
    TaxiEdge*          nextTaxiEdge_p;
};

#endif
```

```
// TaxiEdgeList.h

#ifndef _TaxiEdgeLIST_H
#define _TaxiEdgeLIST_H

#include "TaxiEdge.h"

class TaxiEdgeList{
public:
    TaxiEdgeList();
    TaxiEdgeList(TaxiEdge* TaxiEdge_p);
    ~TaxiEdgeList();
    void append(Edge* const);                       // Append ELEM at tail_p of list
    void setFirst();                                // set curr_p to first position
    void moveNext();                                // Move curr_p to next position
    void movePrev();                                // Move curr_p to previous position
    int  length() const;                            // Return current length of list
    Edge* currEdge_p() const;                       // Return value of current ELEM
    Edge* nextEdge_p() const;                       // Return value of next ELEM
    Edge* tailEdge_p() const;                       // Return value of tail ELEM
    bool isInList() const;                          // TRUE if curr is within list
    TaxiEdge* read_curr_p();
    void write_curr_p(TaxiEdge* cTaxiEdge_p);
    void print_taxiEdgeList(TaxiEdge* cTaxiEdge_p);
private:
    TaxiEdge *head_p,
              *tail_p,
              *curr_p;
};

#endif
```

```
// TaxiPath.h
#ifndef _TAXIPATH_H
#define _TAXIPATH_H
#include "Edge.h"

class TaxiPath{
public:
    TaxiPath();
    TaxiPath(Edge* currEdge_p);
    ~TaxiPath();
    Edge*              read_edgeEle_p();
    TaxiPath*          read_nextTaxiPath_p();
    void write_nextTaxiPath_p(TaxiPath* tempNextTaxiPath_p);
private:
    Edge*              edgeEle_p;
    TaxiPath*          nextTaxiPath_p;
};

#endif


// TaxiPathList.h

#ifndef _TAXIPATHLIST_H
#define _TAXIPATHLIST_H
#include "TaxiPath.h"
class TaxiPathList{
public:
    TaxiPathList();
    TaxiPathList(TaxiPath* taxiPath_p);
    ~TaxiPathList();
    void append(const TaxiPath&);                  // Insert ELEM at tail_p of list
//  TaxiPath* remove();                            // Remove and return current ELEM
//  void setFirst();                               // set curr_p to first position
    void next();                                   // Move curr_p to next position
    void prev();                                   // Move curr_p to previous position
    int  length() const;                           // Return current length of list
//  int setPos(const int);                         // Set curr_p to specific position
//  void setValue(const TaxiPath&);                // Set current ELEM's value
    Edge* currEdge_p() const;                      // Return value of current ELEM
    Edge* nextEdge_p() const;                      // Return value of next ELEM
//  bool isEmpty() const;                          // TRUE if list is empty
    bool isInList() const;                         // TRUE if curr is within list
//  bool find(const TaxiPath&);                    // Find the specifin value
private:
    TaxiPath *head_p,
             *tail_p,
             *curr_p;
};

#endif
```

```
//Type.h
#ifndef _TYPE_H
#define _TYPE_H

enum Node_Type{
            taxiNode,
            rwyNode,
            gate,
            rwyIntersectionNode
};
enum Edge_Type{
            taxiEdge,
            rwyEdge,
            exitEdge
};
enum Operation_Type{
            departure,
            arrival
};
//enum Permission_Type          {yes, no};
enum VisitMark_Type    {visited, unvisited};
enum Priority_Type     {low, high};                // flight's priority for conflicting at intersection
enum ConflictChecked_Type       {done, notYet};             //   flight's   priority   for   conflicting   at
intersection
enum CurrentEdgeDirection_Type           {thisDirection, reverseDirection};    //        Edge's        curretn
operational direction
enum Aircraft_Type     {heavy, large, smal_};       // "small" is reserved for certain compiler
enum TwoOperations{dep_dep, dep_arr, arr_dep, arr_arr};

enum SP_Algorithm{
            static_SP,         // static shortest path algorithm 1
            td_SP_1,  // time-dependent shortest path algorithm 1
            td_SP_2            // time-dependent shortest path algorithm 2
};

enum FlightState_Type{
     //for departure
              pushingBack,               // taxing from gate to arpon(=ramp area)
              rolling,
              liftingOff,
              endOfDeparture,   // take-off (rolling with const accel)
              waitingOnRunway,  // waiting around the runway or gate or from runway exit to gate
     //for arrival
              onFinal,
              flaringOut,
              freeRolling,
              braking,
              coasting,
              adjustingBrake,
              freeRollingSlow,
              exitingRunway,
    //touch-down -> exit
    //for taxing for both operations (arrival/departure)
              parking,
              taxingToDepartureQueue,
              taxingToGate,
              delayedAtCross, //
              waitingToTaxi,
              speedUp,
              steadyTaxing,
              slowDown,
              turning,          // taxing from gate to the around of runway
              waitingInLine,
              areaHolding,
    //for both operations' communication
              waitNextContact,
              readyToCommunicate,
              sendingRequest,
              waitingCommand,
              receivingCommand,
              sendingConfirmation,
              waitControllerContact
};
enum IdleBusy_Type     {idle, busy};
enum ControllerState_Type{
              standby,
              receivingRequest,
              judgingCommand,
              sendingCommand,
              receivingConfirmation
};
```

```
enum ControllerMessage_Type{
              unDecided,
              fileApproved,
              reRoute,
              stopThere,
              waitThere,
              //waitControllerContact,
              reStart,
              clearToPushback,
              clearToTaxi,
              clearToTakeOff,
              clearToLand
};
enum PilotRequest_Type{
              requestTaxi,
              requestPushBack,
              requestTakeOff,
              requestLanding
};
#endif
```

VITA

Hojong Baik was born in Seoul, Korea on May 1, 1962. After completing Dongsung High School in Seoul, he entered the Civil Engineering program of Yonsei University in 1981. Upon completion of his bachelor degree, he entered the graduate school in Seoul National University for his master degree.

In May 1989, after graduating with his master degree in transportation planning and completing his military service in the army, he joined the Korea Transport Institute (KOTI) as a researcher. He had worked at KOTI for more than five years in KOTI. During this period, he joined several projects in various fields including urban transportation, intelligent transportation system (ITS), and logistics.

In August 1994, he enrolled the Department of Civil Engineering at Virginia Polytechnic Institute and State University, to pursue his Ph. D. degree. He worked as a teaching assistant and also as a research assistant at the National Center of Excellence for Aviation Operations Research (NEXSTOR).