2003

A Heuristic Method for a Rostering Problem with the Objective of Equal Accumulated Flying Time

Xugang Ye

THE FLORIDA STATE UNIVERSITY
COLLEGE OF ARTS AND SCIENCES

A Heuristic Method for A Rostering Problem with The Objective of Equal Accumulated Flying Time

By

Xugang Ye

A Thesis Submitted to the
Department of Mathematics
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
Fall Semester, 2003

The members of the committee approve the thesis of Xugang Ye defended on June 19, 2003.

Steve Blumsack
Professor directing thesis

Steve Bellenot
Committee Member

Robert N. Braswell
Committee Member

The Office of Graduate Studies has verified and approved the above named committee members

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to Dr. Steve Blumsack for the opportunity to explore the field of Combinatorial Optimization. His encouragement, guidance, and support were invaluable in my work.

I would also like to thank Dr. Steve Bellenot and Dr.Robert N. Braswell for their support as my professors and committee members.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Next to fuel costs, crew costs are the largest direct operating cost of airlines. Therefore much research has been devoted to the planning and scheduling of crews over the last thirty years. The planning and scheduling of crews is a highly complex combinatorial problem that consists two independent phases. The first phase is the Crew Pairing Problem (CPP), which concerns finding a set of tasks with minimum cost while satisfying the service requirements. The second phase is the Crew Rostering Problem (CRP), which concerns finding work assignment for crewmembers in a given period.

In this thesis we focus on a Crew Rostering Problem, where a main pilot and a copilot perform a task. The model is a variance minimization problem with 0-1 variables and constraints associated with ensuring collective agreements, rules and guaranteeing the production of flights service. We choose a sequential constructive method (heuristic) to solve this difficult combinatorial problem since: (1), minimizing quadratic function of discrete variables makes linear methods difficult to use, a monthly schedule for one hundred pilots can generate tens of thousands variables and millions of constraints; (2), it is a NP-hard problem, which means the CPU time of solution searching will grow exponentially as the instance dimension (the number of pilots and the number of tasks) increases. According to the characteristics of the model we propose, we do not find the global optimal solution; we find a satisfactory solution (or near optimal solution).

The basic idea in our heuristic method is to decompose the assigning process into many subphases day by day. Then in dealing with minimizing the objective function, two

heristic principals are employed. Meanwhile, in coping with the constraints, a weighted matching model and its algorithm will be used. In the numerical simulation, the comprehensive method is tested for its effectiveness. We show that our method can produce a solution whose objective value is below a satisfactory bound.

CHAPTER 1

INTRODUCTION TO THE PROBLEM

INTRODUCTION

In today's highly competitive markets, computer-aided systems for crew planning and scheduling have become a critical success factor of the airline industry. Costs to operate a timetable of tasks can be substantially reduced by finding the most efficient use of company's expensive crew resources, using OR (Operation Research) techniques. For increasing size of timetables, real-life crew scheduling problems become more and more complicated. This is a source of new challenges for mathematical programming.

Crew planning and scheduling has received a lot of attention in the Operation Research literature. Yet, only very recently, cases are reported where companies in the airline industry are using advanced OR techniques for solving crew planning and scheduling problems (almost) optimally [2]. With the rapidly increasing computer power in the past decade, advanced OR techniques such as column generation [5] are gradually becoming more and more applicable to real life crew planning and scheduling problems (see e.g. Day and Ryan (1997), Andersson et al. (1998), Desrosiers et al. (2000), and Ryan (2000)). Crew planning and scheduling occurs on several levels, depending on the length of the planning and scheduling period and whether the planning and scheduling is for strategic,

tactical or operational purposes. The two most widely applied crew planning and scheduling problems are the crew pairing problem (CPP) for grouping flights into tasks (or pairings, or duties), and the crew rostering problem for assigning tasks to weekly, monthly or seasonal rosters for individual crew members.

This thesis will investigate a crew rostering problem from Air China. Since its establishment, Air China has more than 40 years history of operation safety due to its advanced aircrafts, aviation system and well-trained pilots. But automation of crew management, especially crew planning and scheduling is only a development in recent couple of years [23]. Although Air China has successfully initiated the computer-assisted system such as Pilot Management and Scheduling (PMS), OR techniques need to be developed and integrated into the system. Unlike the airline companies of other Countries, Air china has its own style of crew management leading to some distinctive models. Instead of general set partition problem [34,35], this thesis introduces a variance minimization model.

In this chapter, we will first discuss briefly the concept of two basic crew planning and scheduling problems, then focus on the rostering problem and formulate our model.

1.1 CREW PAIRING PROBLEM

The crew pairing problem (CPP) is formally defined as follows [14,15]: given a set of flights with fixed starting and ending times and locations, and given a set of *rules* and *criteria*, find a *set of tasks with minimum cost* such that each flight is included in exactly one task and all rules are satisfied. The entities of the problem are as follows: A flight (or flight leg) is a nonstop, inseparable work duration from one airport to another airport in a timetable. Each duty consists of a sequence of flights that satisfy certain rules like

maximum work time, minimum rest time, etc. An example of a flight is Detroit 9:10am - Memphis 10:50am; an example of a duty is Detroit 8:30 am- Detroit 17:00pm. A duty may also cover more than one day, which is then often called a *task*, or *pairing or trip,* here we use task to denote a duty longer than one day. We should notice that in timetables of most airline companies, a task starts and ends at the same crew base. This is what we called *single home base model* [3,11,43]. A Roster, or Timetable for crews, is a set of tasks (and other activities) assigned to a pool of specific crewmembers. Figure 1.1.1 shows the hierarchy relation of the entities in crew pairing:

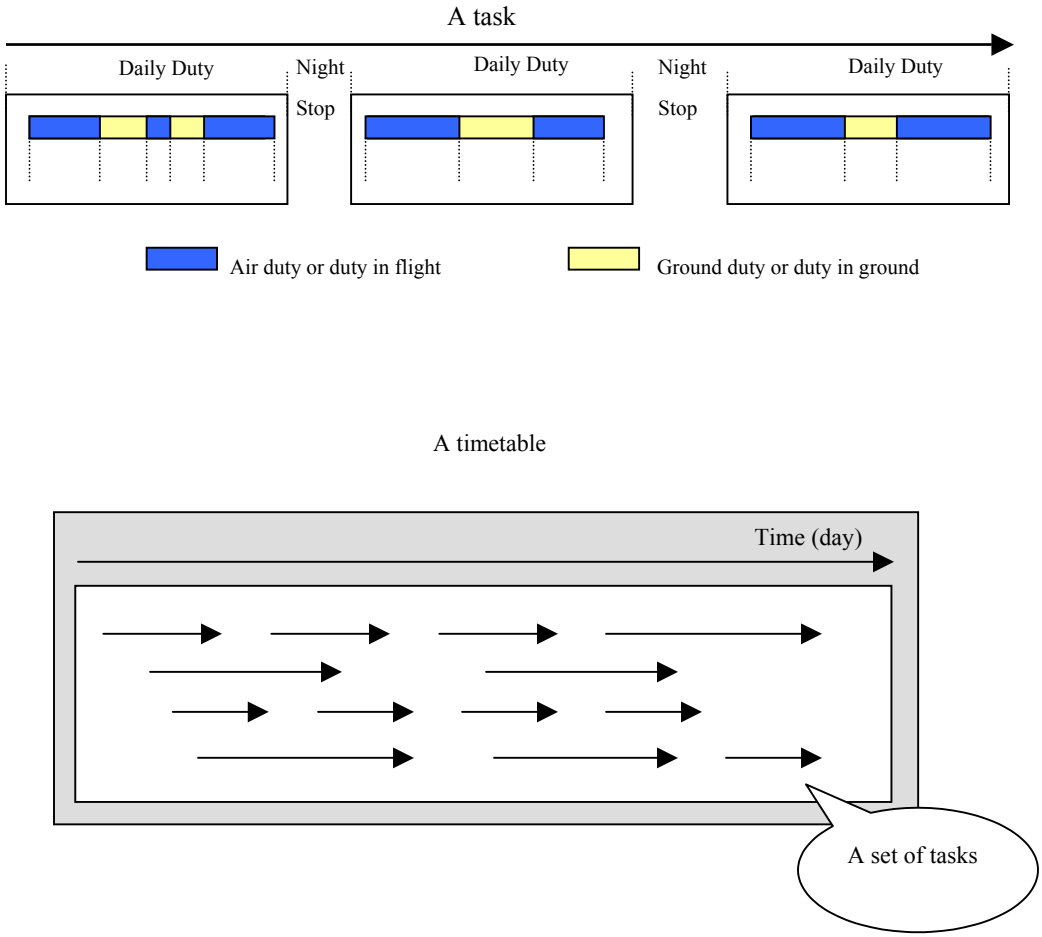

Figure 1.1.1-Basic entities of CPP

Generally, the generation of a duty should take into consideration at least the following constraints [43]: (1) the destination of a flight should be as same as the start place of the immediately next flight; (2) within a duty there is only one type of aircraft; (3) the number of flights within a duty lies between one and a preassumed upper limit; (4) there is a preassumed upper limit for the maximum flying time within a duty; (5) there is a preassumed lower limit for the minimum stop time between two consecutive flights. A duty time is the elapsed time from the duty beginning to the duty end; it includes the real flying time and the ground duty time between the flights

Similarly, the generation of a task (pairing of trip) should take into consideration at least the following constraints [43]: (1) a task starts and ends at the same crew base; (2) the destination of a duty should be as same as the start place of the immediately next duty; (3) within a task, there is only one type of aircraft; (4) the number of duties within a task lies between one and a preassumed upper limit; (5) there is a preassumed upper limit for the overall duty time within a task; (6) there is a preassumed upper limit for the duty time within 24 hours; (7) there is a preassumed lower limit for the minimum rest time between two consecutive duties; (8) The duration of a rest immediately following a duty has a minimum and maximum associated with this duty.

A set of flights is constructed according to the origin destination flow (ODF) of the potential passengers [45]. The set of duties is built on connecting the flights [2]. Then is the further work of connecting duties, which means building tasks, or timetable for crews. Every step of construction follows such a way that as the same time all rules and constraints are satisfied, some optimal objectives are pursued [2].

There will be a large number of feasible tasks for a relatively small fleet. Vance et al. (1997) found that a fleet with 250 daily legs had over 5,000,000 tasks. Larger fleets have

billions of legal tasks. The enormous number of tasks is a major difficulty in solving airline crew scheduling problems exactly. Crew pairing problems are solved by generating possible sets of legal tasks and selecting the optimal set of tasks. This is column generation technique [5]. Recent work on deterministic airline crew pairing include Lavoie et al. 1988, Gershkoff 1989, Anbil et al. (1991, 1992a, 1992b, 1999), Graves et al. 1993, Hoffman and Padberg 1993, Barnhart et al. 1994, Andersson et al. 1997, Chu et al. 1997, Van e et al. 1997, Klabjan et al. (1999a, 1999b, 1999) and Snowden et al. 2000. Yen 2000 also considers the problem of crew scheduling under uncertainty.

In this thesis, we assume that we already solved the crew pairing problem. We will focus on roster construction or crew rostering problem.

1.2 CREW ROSTERING PROBLEM

The crew rostering problem (CRP) is formally defined as follows [43,44]: given the same information as for the CPP, and given a set of crew members with certain *characteristics*, find a *roster* such that each task is included in the roster, all rules are satisfied, and the characteristics of each crew member are taken into account. Examples of such characteristics are qualifications, pre-assigned tasks, individual requests, and the past rosters for the crewmember. The crew's past is necessary for checking laws and optimizing criteria that extend beyond the planning period.

Figure 1.2 [45] shows the relation between the two problems: Crew pairing and crew rostering:

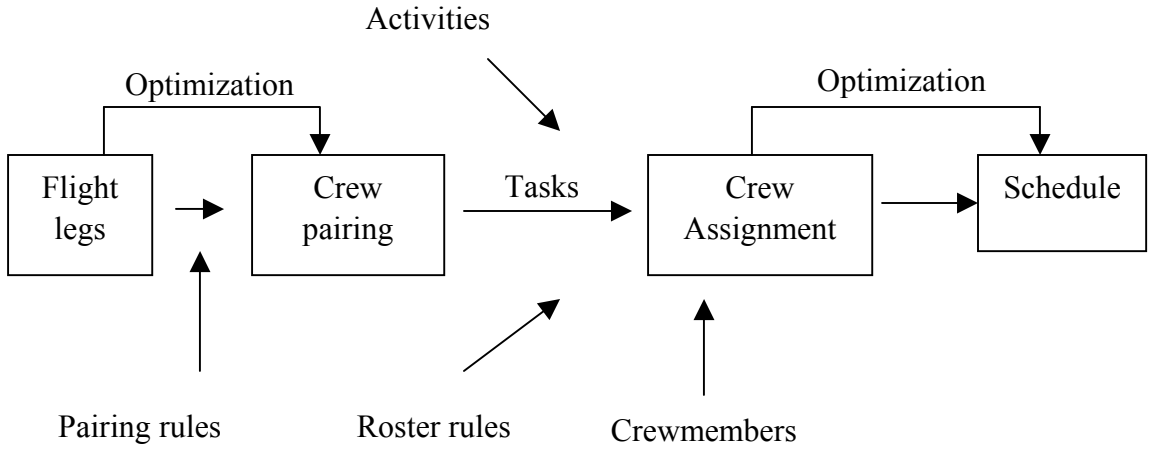

Figure 1.2.1-The relation between CPP and CRP

Crew pairing and rostering problems are usually modeled with set partitioning type of formulations (see Grafinkel and Nemhauser (1969) for classical set partitioning models), and solved with column generation techniques [5]. Many papers have already been published on column generation techniques for crew planning, see e.g. Desaulniers et al. (1997) and Vance et al. (1997) for recent papers on crew pairing, and Gamache et al. (1999) for a recent paper on crew rostering. We will show that in this thesis our problem can also be transformed to a classical set partitioning problem when some simplifications are made. Also, the heuristic algorithm we propose is a form of column generation technique because in constructing roster, we select the optimal set of tasks from some possible sets of legal tasks by two heuristic principles.

In most literature, the essence of crew rostering problem is to construct personalized schedules for pilots [30,43,44]. Personal events and personal preference should be accommodated into the overall constraints. Generally events such as training or maintenance can be added as visual tasks [8,43]. Personal preference leads to more additional constraints in addition to those constraints stemming from pilot rank, class, or license [8,44].

Possible objective functions include the minimization of the number of personalized schedule [8,45], maximization of the satisfaction of personal preference [8] etc. Generally, biweekly, monthly, or seasonal personalized schedules are constructed. Personal events are dealt with simultaneously with tasks assignments in a generalized timetable [43,44,45].
In this thesis, we consider a simple case without individual requirements from pilots. So, we only consider the tasks assignment. Even the most simple rostering problem with dozens of pilots and a biweekly timetable has thousands of variables and tens of thousands of constraints. So crew rostering problem is generally viewed as a huge interger programming [3,34,35]. Since exact algorithms (for example, branch and bound) are not time efficient, most existing methods, no matter what the objective functions may be, use sequential heuristics to decompose the feasible solution space to decrease the computational scale [4,6,42].

In this thesis a day-by-day sequential heuristic will be employed for the model proposed. Now, before introducing the problem that will be investigated in this thesis, we give a brief introduction to the data management of the assigning process of our roster construction.

1.3 DATA MANAGEMENT

The assigning process is a dynamic, state-dependent process; it is implemented by the update of sets of structured data. At present, the Data Base Management System (DBMS) has been widely and successfully applied to the management and operation of the airline industry [14,25,30]. It is the data source of the OR-based Decision-making Support System (DSS). Moreover, there exists the dynamic data interaction between DBMS and DSS. The technical development of the management and operation of the airline industry is intensively embodied by the development of DBMS and DSS [23,36]. As follows, we will put forward what data we need to play with and where our mathematical model is formulated. First is the data from pilots.

Pilots are the people who put all sophisticated equipment to work. On most commercial airlines, there are always at least two pilots, and on many flights, there are three.

On an airliner, the pilot in command is called the captain or main pilot. The captain, who generally sits on the left side of the cockpit, is ultimately responsible for everything that happens on the flight. This includes making major command decisions, leading the crew team, managing emergencies and handling particularly troublesome passengers. The captain also flies the plane for much of the trip, but generally trades off with the first officer at some point. Because a main pilot takes most of the responsibilities in a task, generally he (she) has the privilege of being considered in prefernce in assignments. In this thesis we also use a main-pilot-preferred principle in our heuristic algorithm.

The first officer, or copilot, who is the second in command, sits on the right side of the cockpit. He or she has all of the same controls as the captain, and has had the same level of training. The primary reason for having two pilots on every flight is safety. Additionally, the first officer provides a second opinion on piloting decisions, keeping pilot error to a minimum.

Most airliners built before 1980 have a cockpit position for a flight engineer, also called the second officer. Typically, flight engineers are fully trained pilots, but on an ordinary trip, they don't fly the plane. Instead, they monitor the airplane's instruments and calculate figures such as ideal takeoff and landing speed, power settings and fuel management. In newer airliners, most of this work is done by computerized systems, eliminating the need for the flight-engineer position. In the future, it will be phased out entirely.

The problem in this thesis only considers first two pilots: captain, or main pilot and first officer, or copilot. The problem considers the matching between the main pilot and the copilot. It also considers how to balance the accumulated flying time among the main pilots group and the copilots group respectively.

Sometimes, the classification of pilot rank is not so strict as the licenses show. Some main pilots may be assigned copilot duties; some excellent copilots may be assigned captain duties. It depends on whether or not the number of pilots is enough to cover the considered set of the tasks [23,36,43]. In Air China, the ratio of pilots vs. aircrafts is much higher than Airline Companies in North America and western Europe [23]. So we assume the number of both main pilots and copilots is sufficient for covering the considered set of tasks.

The Figure 1.3.1[23] describes the data interaction between the DBMS and the DSS. We can see that DBMS is the basic data source; the three tables Table1.1 to Table 1.3 are the data view that describes the data we need for the input port of the model, it is the interface between the DBMS and the DSS. Our mathematical model and its algorithm are the core techniques of the Tasks assigning process DSS.

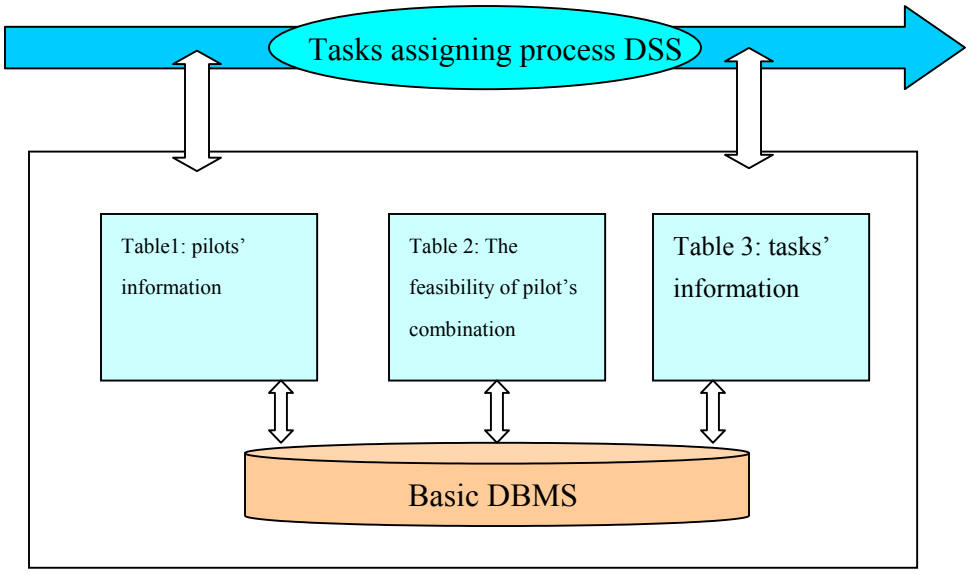

Figure 1.3.1-Data interaction between DBMS and DSS

Table 1.3.1:Pilots' information

ID	Name	rank	Accumulate flying time (minutes)						
			Date: August 2001						
			1	2	3	……	29	30	31
00325	John	Main pilot	3077	?	?	……	?	?	?
00137	Mason	Copilot	2451	?	?	……	?	?	?
……									
00216	Lynch	Copilot	1159	?	?	……	?	?	?
00244	Dodge	Main pilot	946	?	?		?	?	?

Table 1.3.2: The feasibility of pilot's combination

ID		copilots			
		0013	0054	……	0129
Main pilots	0048	1	0	……	0
	0037	0	0	……	0
	……	……		……	……
	0115	0	1	……	0

Note: The 0-1 matrix tells if a main pilot and a copilot can be paired together. "1" means no, "0" means yes.

Table 1.3.3: tasks' information

Task No.	Task components	Start time	End time	TaskTime (minutes) (end - start)	Flying time (minutes)	Main pilot & copilot
0001	D-N-L -C-D	2001-8-1-7:00	2001-8-3-9:17	3017	604	?
0002	D-N-D -C-D	2001-8-1-7:30	2001-8-3-8:14	2924	521	?
……	……	……	……	……	……	……
0527	D-M-D -M-D	2001-8-31-22:30	2001-9-4-19:10	4120	805	?

1.4 PROBLEM AND MATHEMATICAL FORMULATION

Suppose in a period of time we want to assign a set of tasks to a pool of pilots. Suppose our pilots consist of those ranked as main pilots and those ranked as copilots, a main pilot and a copilot cooperate to accomplish a task. Because there are some regulations on the combination of the pilots in a task, some main pilots cannot be assigned together with some copilots. We want to find an assignment such that the each main pilot's accumulated flying time is as close to any other main pilot's accumulated flying time as

possible; each copilot's accumulated flying time is as close to any other copilot's accumulated flying time as possible.

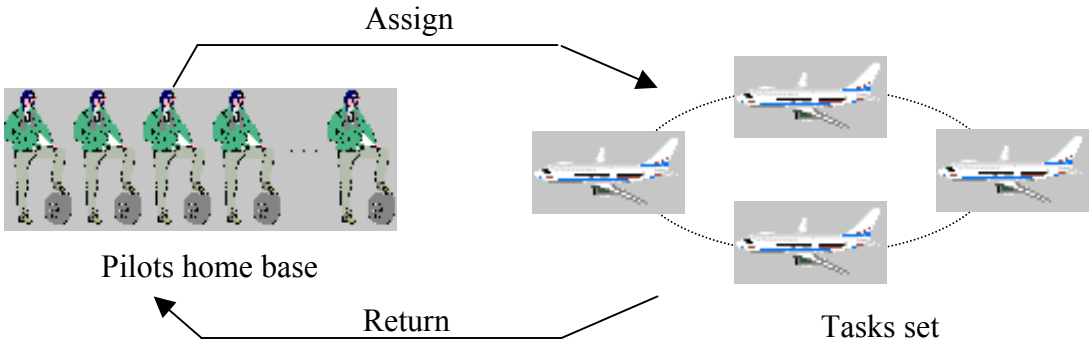

Figure 1.4.1- Rotation rostering

In a period of time suppose we have n tasks: $TASK_1$, $TASK_2$, …, $TASK_n$, we have m_1 main pilots, and m_2 copilots.

Let x_{ik} $(i=1,2,\cdots,m_1;\ k=1,2,\cdots,n)$ be *0-1* integer variables.

$$x_{ik} = \begin{cases} 1 & \text{if the ith main pilot will be assigned the kth task;} \\ 0 & \text{if the ith main pilot will not be assigned the kth task} \end{cases}$$

Let y_{jk} $(j=1,2,\cdots,m_2;\ k=1,2,\cdots,n)$ be *0-1* integer variables.

$$y_{jk} = \begin{cases} 1 & \text{if the jth copilot will be assigned the kth task;} \\ 0 & \text{if the jth copilot will not be assigned the kth task} \end{cases}$$

Let parameters: B_k $(k=1,2,\cdots,n)$, E_k $(k=1,2,\cdots,n)$, and T_k $(k=1,2,\cdots,n)$,be the begin time, end time and flying time (minutes, positive integer) of $TASK_k$ (k=1,2,⋯,n), respectively. Without losing generality, let:

$$B_1 < B_2 < \cdots < B_n .$$

Let parameters: E^T_i $(i=1,2,\cdots, m_1)$ be the end time of the task the ith main pilot takes immediately before he can take his first task of this period; Let parameters: E^t_j $(j=1,2,\cdots, m_2)$ be the end time of the task the jth copilot takes immediately before he can take his first task of this period.

Let z_i $(i=1,2,\cdots,m_1)$ be the accumulated flying time of the ith main pilot. Let $z_i^{(0)}(i=1,2,\cdots,m_1)$ be the initial accumulated flying time of the ith main pilot. Let f_j $(j=1,2,\cdots,m_2)$ be the accumulated flying time of the jth copilot. Let $f_j^{(0)}(j=1,2,\cdots,m_2)$ be the initial accumulated flying time of the jth copilot.

Parameter p_{ij} $(i=1,2,\cdots,m_1; j=1,2,\cdots, m_2)$ tells if we can pair the ith main pilot and the jth copilot together for a same task, that is:

$$p_{ij} = \begin{cases} 0 & \text{if the ith main pilot can be paired together with the jth copilot;} \\ 1 & \text{if the ith main pilot can not be paired together with the jth copilot} \end{cases}$$

The mathematical model of this problem can be formulated as following (1.1-1.12).

At first glimpse, without the constraints set (1.12), there would be two seperate variance minimization problems. Each one is a complicated 0-1 nonlinear combinatorial optimization problem.

$$\min \quad (s_1 - s_1^*)^2 + (s_2 - s_2^*)^2 \tag{1.1}$$

$$s_1^2 = \frac{1}{m_1}\sum_{i=1}^{m_1}(z_i - \frac{1}{m_1}\sum_{i=1}^{m_1} z_i)^2, s_1 \geq 0 \tag{1.2}$$

$$\begin{cases} z_i = z_i^{(0)} + \sum_{k=1}^{n} x_{ik} T_k & i = 1,2,\cdots,m_1 & (1.3) \\ B_k \geq x_{ik} \cdot E_i^T & i = 1,2,\cdots,m_1, k = 1,2,\cdots,n; & (1.4) \\ \sum_{i=1}^{m_1} x_{ik} = 1 & k = 1,2,\cdots,n; & (1.5) \\ (x_{iq} + x_{ik} - 1) \cdot E_k \leq B_q & i = 1,2,\cdots,m_1, & (1.6) \\ & k,q = 1,2,\cdots,n, \quad k < q; & \end{cases}$$

$$s_2^2 = \frac{1}{m_2}\sum_{j=1}^{m_2}(f_j - \frac{1}{m_2}\sum_{j=1}^{m_2} f_j)^2, s_2 \geq 0 \tag{1.7}$$

$$\begin{cases} f_j = f_j^{(0)} + \sum_{k=1}^{n} y_{jk} T_k & j = 1,2,\cdots,m_2; & (1.8) \\ B_k \geq y_{jk} \cdot E_j^t & j = 1,2,\cdots,m_2, k = 1,2,\cdots,n; & (1.9) \\ \sum_{j=1}^{m_2} y_{jk} = 1 & k = 1,2,\cdots,n; & (1.10) \\ (y_{jq} + y_{jk} - 1) \cdot E_k \leq B_q & j = 1,2,\cdots,m_2; & (1.11) \\ & k,q = 1,2,\cdots,n, \quad k < q; & \end{cases}$$

$$x_{ik} + y_{jk} + p_{ij} \leq 2 \qquad i = 1,2,\cdots,m_1, j = 1,2,\cdots,m_2, \quad k = 1,2,\cdots,n. \tag{1.12}$$

Where, s_1^* is the optimal solution to problem (1.2-1.6) minimizing (1.2); s_2^* is the optimal solution to problem (1.7-1.11) minimizing (1.7).

The dimension of any instance of problem (1.1-1.12) can be described by a vector *(m_1, m_2, n)*. The number of the variables is $m_1 n + m_2 n$, the number of constraints is: $m_1 + m_2 + (m_1 + m_2)n + 2n + (m_1 + m_2) \cdot \frac{1}{2} n(n-1) + m_1 m_2 n$. For example, let m_1*=58, *m_2*=62, n=200*, then, the number of the variables is $m_1 n + m_2 n$ *=24000*; the number of constraints is: $m_1 + m_2 + (m_1 + m_2)n + 2n + (m_1 + m_2) \cdot \frac{1}{2} n(n-1) + m_1 m_2 n$ = 3131720. Let's give a rough estimation of the size of the feasible solution space: consider the main pilots group only, suppose each pilot can perform at least *[200/58]=3* tasks of the *n=200 tasks*, there will be more than $\binom{200}{3} \times \binom{197}{3} \times \cdots \times \binom{5}{3}$ options! So, if we do not choose an efficeient algorithm, it will be a disaster of combinatorial explosion!

In Chapter 4 we will prove the NP-hardness of the problem (1.1-1.12), the problem(1.2-1.6) minimizing (1.2), and the problem (1.7-1.11) minimizing (1.7). This greatly strengthens our motivation of using heuristic algorithm. Before we propose a heuristic method for this seemly intractable combinatorial optimization problem, let's first turn to some preliminaries on combinatorial optimization and heuristic techniques, which will be introduced in the following chapters.

CHAPTER 2

COMBINATORIAL OPTIMIZATION AND COMPUTATIONAL COMPLEXITY

INTRODUCTION

Combinatorial optimization problems are concerned with the efficient allocation of limited resources to meet desired objectives when the values of some or all of the variables are restricted to be integral. Constraints on basic resources, such as labor, supplies, or capital restrict the possible alternatives that are considered feasible. Still, in most such problems, there are many possible alternatives to consider and one overall goal determines which of these alternatives is best. For example, most airlines need to determine crew schedules which minimize the total operating cost; automotive manufacturers may want to determine the design of a fleet of cars which will maximize their share of the market; a flexible manufacturing facility needs to schedule the production for a plant without having much advance notice as to what parts will need to be produced that day. In today's changing and competitive industrial environment the difference between using a quickly derived "solution" and using sophisticated mathematical models to find an optimal solution can determine whether or not a company survives.

The ubiquity of the combinatorial optimization model stems from the fact that in many practical problems, activities and resources, such as machines, airplanes and people, are

indivisible. Also, many problems have only a finite number of alternative choices and consequently can appropriately be formulated as combinatorial optimization problems -- the word combinatorial referring to the fact that only a finite number of alternative feasible solutions exists. Combinatorial optimization models are often referred to as integer programming models where programming refers to "planning" so that these are models used in planning where some or all of the decisions can take on only a finite number of alternative possibilities.

Combinatorial optimization is the process of finding one or more best (optimal) solutions in a well defined discrete problem space. Such problems occur in almost all fields of management (e.g. finance, marketing, production, scheduling, inventory control, facility location and layout, data-base management), as well as in many engineering disciplines (e.g. optimal design of waterways or bridges, VLSI-circuitry design and testing, the layout of circuits to minimize the area dedicated to wires, design and analysis of data networks, solid-waste management, determination of ground states of spin-glasses, determination of minimum energy states for alloy construction, energy resource-planning models, logistics of electrical power generation and transport, the scheduling of lines in flexible manufacturing facilities, and problems in crystallography). A survey of related applications of combinatorial optimization is given by Grötschel (1992) [31]. Although some research has centered on approaches to problems where some or all of the functions are nonlinear, most of the research to date covers only the linear case [33]. A survey of nonlinear integer programming approaches is given by Cooper and Farhangian (1985) [31]. In this thesis, as our proposed problem in chapter 1 shows, we will investigate how to deal with a nonlinear large scale 0-1 integer programming.

Combinatorial problems occur in many practical applications and there is an urgent need for algorithms, which solve them efficiently. The algorithmic approaches to

combinatorial optimization problems can be classified as either exact or approximate. Exact algorithms are guaranteed to find an optimal solution in finite time by systematically searching the solution space. Yet, due to the NP-completeness of many combinatorial optimization problems, the time needed to solve them may grow exponentially in the worst case. To practically solve these problems, one often has to be satisfied with finding good, approximately optimal solutions in reasonable, that is, polynomial time. This is the goal of approximate algorithms such as local search or solution construction algorithms. Approximate algorithms cannot guarantee optimality of the solutions they return, but empirically they have often been shown to return good solutions in short computation time. For our problem, due to its NP-hardness, a heuristic approximate algorithm will be constructed, before the detailed explanation as to why heuristics work, we need at first have a review of the concepts of combinatorial optimization and the related computational complexity theory.

2.1 COMBINATORIAL OPTIMIZATION PROBLEM

Combinatorial problems are intriguing because they are easy to state but often very difficult to solve. We now elaborate on five specific illustrations of combinatorial problems of practical interest, with an associated reference [31,33] for more details:

1. The travelling salesman problem – An individual has to carry out a tour among n cities, visiting each of them precisely once. Knowing the distances of each direct link between pairs of cities, the objective is to seek the tour (sequence of cities) so as to minimize the total distance travelled.

2. The quadratic assignment problem – For a set of n facilities we know the volume of material (a_{ik}) to be moved per unit time between each pair of facilities (i and k). There is a grid of points on which the non-overlapping facilities can be located. The objective is to position the facilities so as to minimize the total volume and distance of the material movement. The usual modeling approach is to have 0-1 variables:

$$x_{ij} = \begin{cases} 1 & \text{if facility i is located at point j} \\ 0 & \text{otherwise} \end{cases}$$

and the objective function is given by $\sum_i \sum_j \sum_k \sum_l x_{ij} x_{kl} a_{ik} d_{jl}$ where d_{jl} represents the distance between points *j* and *l*. The adjective "quadratic" is used because of the cross products of decision variables in the objective function.

3. The resource-constrained project-scheduling problem – A project is made up of n distinct activities. Each has a given duration and requires the use of one or more resources where there is an upper limit on the availability of each resource. There are also precedence constraints among the activities. The objective is to minimize the completion time of the project.

4. The fixed-charge capacitated multicommodity network design problem – A set of nodes are prescribed with given transportation needs per unit time of different commodities between each pair of nodes. Direct links are possible between each pair of nodes and there is a capacity, a fixed cost and variable costs specified for each such link. The objective is to select the set of links, satisfying the flow requirements, with minimum overall costs. There are applications in the fields of transportation and telecommunications.

5. The vehicle routing problem – The following description is adapted from Pinedo and Simchi- Levi. Consider a distribution or collection system with a single depot (e.g., a warehouse or school) and n geographically dispersed demand points (e.g., retailers or bus stops). The demand points are numbered arbitrarily from 1 to n. At each demand point there are a number of items (e.g., products or students), which are referred to as the demand and which must be brought to the depot using a fleet of vehicles. There are three types of constraints:

   a) Capacity constraints: an upper bound on the number of units that can be carried by a vehicle.

   b) Distance (or travel time) constraints: a limit on the total distance (or time) travelled by each vehicle and/or a limit on the amount of time an item can be in transit.

   c) Time window constraints: a prespecified earliest and latest pickup or delivery time for each demand point and/or a prespecified time window in which vehicles must reach their final destination.

   The problem is to design a set of routes for the vehicles such that each route starts and ends at the depot, no constraint is violated, and total distance travelled is as small as possible.

A combinatorial optimization problem is either a maximization problem or a minimization problem with an associated set of instances. Without loss of generality we will restrict ourselves in this thesis to minimization problems, because every maximization problem can easily be converted into a minimization problem. In mathematical description, we have:

Definition 2.1.1 [31] An instance of a combinatorial optimization problem is a pair (S, f), where S is the finite set of candidate solutions and $f : S \mapsto R$ is a function which assigns to every $s \in S$ a value $f(s)$. The function value $f(s)$ is also called objective function value. The goal of a combinatorial optimization problem is to find a solution $s_{opt} \in S_{opt} \subset S$ with minimal objective function value, that is: $f(s_{opt}) \leq f(s) \quad \forall s \in S$, where, s_{opt} is called globally optimal solution of (S, f), the set S_{opt} is the set of all globally optimal solutions.

Thus, the term problem refers to the general question to be answered, usually having several parameters or variables with unspecified values. The term instance refers to a problem with specified values for all parameters. In the case of minimization problems, the objective function is also often called cost function and the objective function value is called cost value. We assume, without loss of generality, that the cost function assumes only nonnegative values, that is, $f(s) \geq 0$ for all $s \in S$.

A combinatorial optimization problem can actually be attacked in three different versions [37], which are:

1) Search version: Given an instance (S, f), find an optimal solution, that is, an element $s_{opt} \in S_{opt}$.

2) Evaluation version: Given an instance (S, f), find the optimal objective function value $f(s_{opt})$.

3) Decision version: Given an instance (S, f) and a bound L, decide whether there is a feasible solution $s \in S$ with $f(s) \leq L$.

Clearly, the search version is the most general of these as with the knowledge of an optimal solution, the evaluation version and the decision version are trivially solved. S is called the search space. The finiteness of S suggests that any given instance (S, f) can be solved by enumerating the whole set of solutions and picking the one with minimal cost. Yet, this approach is impractical for many problems because the size of the search space, denoted by $|S|$, grows exponentially with instance size. This is the issue concerned by computational complexity theory, which will be introduced in the following.

2.2 COMPUTATIONAL COMPLEXITY

In general, one is interested in solving combinatorial problems as efficiently as possible where efficient usually means as fast as possible. Hence, an important criterion for the classification of problems is the time the best known algorithms need to find a solution for the given problem.

This issue is addressed by the theory of computational complexity and, in particular, by the theory of NP-completeness. Its main aim is to classify problems according to their difficulty to be solved by any known algorithm. For the classification of problems it has been shown to be useful to address the question regarding problem complexity as a worst-case measure [31,33], that is, the complexity of a problem is determined by the hardest conceivable instance.

The time-complexity of an algorithm is measured by a time-complexity function that gives, depending on the instance size, the maximal run-time for the algorithm to solve an instance. The size of a problem instance reflects the amount of data to encode an instance in a compact form. Often it suffices to have an intuitive understanding of the size of an instance; for example, the size of a TSP instance can be measured by the number of cities in the graph. The time-complexity is typically given in terms of the number of elementary operations like value assignments or comparisons; it is formalized by the $O(\cdot)$ notation. Let f and g be two functions from $N \mapsto N$, then we write $f(n) = O(g(n))$ if there are positive integers c and n_0 such that for all $n > n_0$, $f(n) \leq c \cdot (g(n))$.

An algorithm runs in polynomial time, if the run-time is bounded by a polynomial; if the run-time cannot be bounded by some polynomial, the algorithm is said to be an exponential time algorithm.

In complexity theory, a basic difference is made between efficiently solvable problems (easy problems) and inherently intractable ones (hard problems). Usually, a problem is considered efficiently solvable if a solution can be found in a number of steps bounded by a polynomial of the input size. If the number of steps needed to solve an instance grows super-polynomially, or exponentially, we say that a problem is inherently intractable.

2.2.1 NP–completeness

The theory of NP-completeness formalizes the distinction between easy and hard problems. In general, the theory of NP-completeness is concerned with the decision

version of combinatorial problems. The generality of the conclusions drawn is not very limited by this fact because it is obvious that the optimization version of a problem is not easier to solve than the decision version and if the optimization version of a problem can be solved efficiently. Optimization problems typically have an associated decision problem; for example, in the TSP case, the associated decision version asks whether a tour π with cost bound L exists such that $f(\pi) < L$. The evaluation version of an optimization problem can be solved as a series of decision problems using binary search on the bound L .

The theory of NP–completeness distinguishes between two basic classes of problems. One is the class P of tractable problems.

Definition 2.2.1 The class P is the class of decision problems that can be solved by a polynomial time algorithm.

The class NP can be defined informally in terms of a nondeterministic algorithm. Such an algorithm can be conceived as being composed of a guessing stage and a checking stage. If we are given some instance *I*, in the first stage some solution is guessed. This solution is verified by a deterministic polynomial algorithm in the second stage. The class NP is the class of problems that can be solved by such a nondeterministic algorithm. For the class NP this polynomial-time verifiability of the property for some given solution *s* is essential. The polynomial time verifiability also implies that the guessed solution is of polynomial size.

Definition 2.2.2 The class NP consists of those problems that can be solved by a nondeterministic polynomial-time algorithm.

Any decision problem that can be solved by a deterministic polynomial-time algorithm also can be solved by a nondeterministic polynomial-time algorithm, which is P⊆NP. Yet, polynomial-time nondeterministic algorithms appear more powerful than polynomial-time deterministic algorithms. In fact, a relation between them can be established by the following theorem.

Theorem 2.2.1 If $\prod \subseteq$ NP, then there exists a polynomial p such that $\prod$ can be solved by a deterministic algorithm having time complexity $O(2^{p(n)})$.

Probably the most important open question in theoretical computer science today is whether P = NP? It is widely believed nowadays that P ≠ NP, yet no proof of this conjecture has been found so far [1,31,33].

A problem usually is considered intractable if it is in NP\P. As one cannot show that NP\P is not empty, unless a proof for P not equal NP is found, the theory of NP-completeness focuses on proving results of the weaker form if P ≠ NP, then $\prod \subseteq$ NP\P. One of the key ideas needed for this approach is the notion of polynomial-time reducibility among problems.

Definition 2.2.3 A problem $\prod$ is polynomially-reducible to a problem $\prod$', if a polynomial-time algorithm exists that maps each instance of $\prod$ onto an instance of $\prod$' and that for each instance of $\prod$, "yes" is output if and only if for the corresponding instance of $\prod$' the output of the decision procedure is "yes".

Informally this definition says that if $\prod$ can be polynomially reduced to $\prod$', then problem $\prod$' is at least as difficult to solve as problem $\prod$. Using the notion of polynomial reducibility we can proceed to define the class of NP-complete problems.

Definition 2.2.4 A problem $\prod$ is NP-complete if and only if $\prod \subseteq$ NP and for all $\prod' \subseteq$ NP holds that $\prod'$is polynomially reducible to $\prod$.

The class of NP-complete problems is in some sense the class of the hardest problems in NP.

If a NP-complete problem can be solved by a polynomial time algorithm, then all problems in NP can be solved in polynomial time. Yet, so far for no NP-complete problem a polynomial time algorithm could be found. Thus, if we can prove that a problem $\prod$ is NP-complete common belief suggests that no deterministic polynomial-time algorithm exists and the problem cannot be solved efficiently. The first problem that was shown to be NP-complete is the satisfiability problem of propositional logic.

Theorem 2.2.2 The set partition problem is NP-complete.

Until today a huge bunch of problems have been proved to be NP-complete; among those are the traveling salesman problem, the flow shop problem, the quadratic assignment problem, and many others. In this thesis the problem we are concerned can be reduced to the set partition problem, so, the problem can be proved to be NP-completeness, thus, we want to find near optimal, or satisfactory, solutions. Clearly, the search version is not easier than the associated decision problem. Thus, proving that the decision version of a problem is NP-complete implies that also the search version is hard to solve. Problems that are at least as hard as NP-complete problems but not necessarily element of NP are called NP-hard.

Definition 2.2.5 A problem $\prod$ is NP-hard if and only if for all $\prod' \subseteq$ NP holds that $\prod'$ is polynomially reducible to $\prod$.

Therefore, any NP-complete problem is also NP-hard. On the other side, if the decision version of an optimization problems is NP-complete, the optimization problem is NP-hard.

2.3 SOLUTION METHODS FOR COMBINATORIAL PROBLEMS

Due to the practical importance of combinatorial (optimization) problems, many algorithms for their solution have been devised. These algorithms can be classified as either exact or approximate algorithms. Exact algorithms are guaranteed to solve every finite size instance of a combinatorial optimization problem within an instance-dependent run-time. Yet, due to the inherent complexity of combinatorial optimization problems, many of them are NP-hard, exact methods need exponential run-time in the worst case and we would have, informally speaking, wait for years to get an answer. Therefore, we have to resort to more ad-hoc methods and typically to sacrifice the guarantee of finding optimal solutions for the sake of getting good solutions in polynomial-time using approximate algorithms.

2.3.1 Exact Algorithms

For finite size problems a straightforward exact algorithm is to simply enumerate the full solution space. Yet, such an algorithm is impractical due to the exponential size of the solution space. To increase efficiency, all modern exact methods use pruning rules to discard parts of the search space in which the (optimal) solution cannot be found. These

approaches are doing an implicit enumeration of the search space. For optimization problems the best-known examples are branch & bound algorithms [31], also known as A* in the AI community and dynamic programming. For satisfaction problems most algorithms are improvements over simple backtracking-style algorithms. Examples are algorithms for the solution of CSPs (The Constraint Satisfaction Problem [31]) or the Davis-Logemann-Loveland procedure [31] for the satisfiability problem and its modern variants. Clearly, an advantage of exact methods for satisfaction problems is that they are able to show that instances cannot have solutions. But still, an exact algorithm may not find solutions on satisfiable instances in reasonable time.

For some specific problems, exact algorithms have been improved significantly in recent years and yield impressive results. This is the case, for example, for the Euclidean traveling salesman problem. Using modern branch & cut algorithms, the largest non-trivial instance that has been solved optimally comprises 13509 cities [37]. But, the time needed to find the optimal solution and prove its optimality may take very long time. For example, it is reported in that it took several CPU-years on a network of modern Workstations (Sun Sparc-2) to solve an instance with 7397 nodes[37]. Furthermore, the efficiency of the optimization codes depends very strongly on the characteristics of the instances. There are still much smaller instances of TSPLIB exist for which the optimality could not be proved yet. Despite these successes, on many combinatorial optimization problems the performance of exact algorithms is much less formidable. It is also suggested "the TSP is not a typical combinatorial optimization problem, since most such problems seem significantly hard to solve to optimality" [33]. Such an example is the quadratic assignment problem, for which instances of dimension N>25 are currently beyond the capabilities of state-of-the-art branch & bound algorithms [31,32,33].

2.3.2 Approximate Algorithms

Approximate algorithms differ essentially from exact ones as they cannot guarantee to find optimal solutions in finite time or prove that no solutions exist in the case of satisfaction problems. But, for optimization problems, they often find high quality solutions much faster than exact algorithms and are able to successfully attack large instances. Approximate algorithms can be classified as either constructive or local search algorithms. Additionally, approximate methods may also be obtained by stopping exact methods before completion, for example, after some given time bound.

Constructive algorithms generate solutions from scratch by adding to initially empty solution components in some order until a solution is complete. They are typically the fastest approximate methods, yet they often return solutions of inferior quality when compared to local search algorithms.

The most effective approximate algorithms today are local search algorithms. Local search algorithms start from some initial solution and iteratively try to replace the current solution by a better solution in an appropriately defined neighborhood of the current solution. The most basic local search algorithm is iterative improvement that only replaces the current solution with a better one and stops as soon as no better, neighbored solutions can be found anymore. For a discussion of this algorithm and specific issues for local search, we refer to the next chapter. Often, constructive algorithms are used to generate good initial solutions for the subsequent application of a local search algorithm. For many problems this has been shown to be a promising approach to provide better solutions than when starting the local search from randomly generated solutions.

Local search is not a particularly new method for attacking NP-hard problems. First applications of local search have already been described in the late fifties and the early sixties. Yet, the initial interest in local search algorithms decreased because of the lack of new conceptual work and its success has only been based on its practical usefulness. Additionally, if high solution qualities are required and large problems are to be solved, high computing power is needed which was not available in the early years of computer science.

It is only in the last ten to fifteen years that local search algorithms have become very popular and that they are very successfully applied to many problems. The renewed interest in local search algorithms has several reasons. An important aspect is that local search algorithms are intuitively understandable, flexible, generally easier to implement than exact algorithms, and in practice have shown to be very valuable when trying to solve large instances. The solution of large instances has been made feasible by the development of more sophisticated data structures, for example, to search more efficiently the neighborhood of solutions, and the enormous increase in computer speed and memory availability. Also on the theoretical side considerable progress has been made. One such progress is the treatment of local search from a complexity point of view that lead to a renewed interest of theoreticians in local search algorithms. Additionally, some of the recently developed local search algorithms can be analyzed mathematically; an example is the convergence proofs for simulated annealing [33].

In fact, simulated annealing is a general search scheme that can be applied to many different problems to improve local search performance. Several other such general search schemes have been developed in recent years and many of them are inspired by naturally occurring processes. The inspiring processes have a strong appeal to design new local search paradigms and have led to considerably improved applications of local

search. Among the naturally inspired metaheuristics are simulated annealing [37], evolutionary algorithms [37] (with the main representatives being genetic algorithms, evolution strategies, and evolutionary programming), neural networks [33], and ant colony optimization [32]. These general search schemes are nowadays called metaheuristics. In general, metaheuristics are defined to be an iterative generation process which guides a subordinate heuristic by combining intelligently different concepts of search space exploration and exploitation in order to find efficiently near-optimal solutions. Metaheuristics are designed to be general-purpose algorithms that can be applied without major modifications to many problems.

Apart from the nature-inspired metaheuristics, several others have been devised to intelligently solve hard combinatorial optimization problems without recurring to analogies with natural phenomena. The best known of these are tabu search [33,37], iterated local search algorithm [33,37] and GRASP [37]. In next chapter, we will give a more detailed introduction to the basic types of heuristic methods.

CHAPTER 3

HEURISTIC METHODS

INTRODUCTION

The use of mathematical models to aid in decision making regarding real world situations ends up with a well-defined mathematical representation of a problem, specifically there is an objective or evaluation function that provides the value of any specific solution (values of the set of decision variables) and there are specified constraints that define the region of feasible solutions. Ideally one would like to select the optimal solution that achieves the maximum (or minimum) value. Practical one would like to get the so-called near optimal solution or satisfactory solution in an acceptable, or efficient span of time

While heuristics have been put into widely use in optimization problems, especially combinatorial ones, they are also used in everyday decisions without the associated construction of a mathematical model. Then what is meant by a heuristic (solution) method? There are many possible definitions. We adopt the following, modified slightly, from Foulds [18]. The term heuristic means a method which, on the basis of experience or judgment, seems likely to yield a reasonable solution to a problem but which cannot be guaranteed to produce the mathematically optimal solution.

Zanakis et al [16] for the period 1975-1986 classified, by type of heuristic and area of application, some 442 articles from a selection of surveyed journals. There is a vast

literature related to heuristics. Books, specifically focusing on the topic, include Michalewicz and Fogel, Morton and Pentico, and Reeves [26,32]. Articles of a tutorial nature, some with extensive reference lists, include Foulds, Ignizio, Muller-Malek, Matthys and Nelis, Müller-Mehrbach, Pinedo and Simchi- Levi, Silver, Vidal and de Werra, White, Zanakis and Evans, and Zanakis et al [28].

The following begins with why heuristic methods are used. This is followed by a framework of a variety of basic types of heuristic methods. The subsequent section is devoted to so-called metaheuristics, very general approaches to obtaining solutions of complicated combinatorial problems. Then we comment on the role of interactive methods involving humans and the computer. Next, there is a focus on evaluating the performance of a heuristic.

3.1 WHY USE A HEURISTIC METHOD

There are at least three circumstances that each can lead to heuristics, namely:

1. A combinatorial explosion of the possible values of the decision variables;
2. Difficulty in evaluating the objective function (or in having probabilistic constraints) due to the presence of stochastic variables;
3. Conditions that change markedly with item, the latter requiring a whole time series of solutions, rather than a single solution at a point in time.

To achieve optimality one is forced to simplify the model by introducing questionable assumptions. Moreover, there is the related issue of inaccuracy of the associated needed data. Recognizing these factors it indeed is likely to be better to achieve a reasonable

(non-optimal) solution to a more accurate model than to seek the optimal solution to an incorrect or oversimplified model of the real-world problem. To summarize, heuristics, because they do not require the strictly restrictive assumptions of optimization routines, permit the use of models that are more representative of the real-world problems.

There are often a large number (n) of decision variables, many of which can only take on discrete values (e.g. 0-1 or integer values). This includes the grouping, ordering (permutation), or selection of discrete objects. Even when the objective function is linear (which may not be an appropriate representation) there can be many local optima. Such problems are labelled as nondeterministic-polynomial-time-complete (NP-complete) and all algorithms currently available for finding optimal solutions to them require a number of computational steps that grows exponentially with the size of the problem (two illustrations of what is meant by size are 1, the number of jobs to be scheduled in a machine scheduling problem and 2, the number of facilities to be located with materials to be moved between them). In addition, for some problems it may be extremely difficult to find any feasible solution, let alone the optimal one.

To this point we are devoted to a single, major reason for using heuristics, namely that it is difficult, if not impossible, to obtain the optimal solution of the mathematical model representation of the problem under consideration.

However, there are other reasons for utilizing heuristic solution methods. They include:

Facilitation of implementation – People would rather live with a problem they cannot solve than accept a solution they cannot understand. The acceptance and use by decision makers of decision rules are likely to be facilitated by an, at least intuitive, understanding of how the rules operate, and in particular, how key parameters influence the chosen

actions. This type of understanding is more likely with heuristic rules than with a complex optimization routine[26]. However, this does not necessarily mean that heuristics must always be simple in nature, for some complex problems simple heuristics may not produce acceptable solutions[33].

Show improvement over current practices – related to the previous point, managers may be quite satisfied with a heuristic solution that produces better results than those currently achieved[26].

Fast results – Sometimes fast, reasonable, results are needed and heuristics can be more quickly developed and used than classical optimization routines[33].

Robustness – Heuristics can be less sensitive to variations in problem characteristics and data quality. Optimal solutions are fragile in the sense that they can be exquisitely sensitive to changes in the data[26]. If the problem description changes slightly, to recover an optimal solution generally requires resolving the entire problem (which typically was computationally expensive to solve in the first place). In contrast, heuristics frequently partition the problem and so ignore interrelationships between partitions. This allows updates to be confined to just the partition affected. Recomputation can be local and therefore faster. Moreover, some constraints are actually flexible in practice and a heuristic method can more easily accommodate this flexibility.

Use within optimization routines – Heuristics can be profitably used within optimization routines in three ways. First, they can provide good initial solutions for an iterative scheme. Second, they can furnish bounds to facilitate elimination of portions of the solution space in partial enumeration optimization methods. Third, heuristics can be used to guide the design of search process.

3.2 BASIC TYPES OF HEURISTIC METHODS

This section is devoted to a categorization of basic heuristic methods. It should be pointed out that the categories are not necessarily mutually exclusive. Indeed, it often makes sense to blend more than one type of heuristic in the solution of a specific class of problems. Moreover, it can be fruitful to use two or more distinct methods in parallel to solve the same problem, choosing the best of the solutions. Müller-Mehrbach[28] provides a more general discussion of using combinations of heuristics.

For a given problem, the development of a new heuristic or the choice among existing options is a creative undertaking; hence the following quotation[32] is relevant: "Creativity involves a willingness to break away from established patterns and try new directions, but it does not mean being different for the sake of being different or an exercise in self-indulgence. It is as much a mistake to ignore the accumulated knowledge of the past as it is to be limited by it. Being creative means combining knowledge and imagination." In other words, it makes good sense to be familiar with as much as possible of the existing theory related to the specific or similar mathematical models, as well as with the range of available heuristic approaches. The choice of which heuristic (or metaheuristic) approach to use depends upon a number of factors including:

1. Whether the decision area is strategic, tactical or operational;
2. The frequency with which the decision is made;
3. The development time available;
4. The analytical qualifications of the decision maker(s) involved.
5. The size of the problem (including the number of decision variables),
6. The absence or presence of significant stochastic elements.

3.2.1Randomly Generated Solutions

One relatively straightforward concept is to randomly generate feasible solutions to the problem, evaluate each and choose the best. We can decide on the number of trials so as to achieve a desired probability that the best solution obtained is better than a prescribed percentage of all solutions. We can also look at biasing the sampling, including adapting the biasing as results are observed. This is closely linked to one of the metaheuristics, the adaptive reasoning technique, to be discussed later.

3.2.2 Problem Decomposition / Partitioning

Here we take a complex problem and decompose or partition it into a number of, presumably simpler to solve, subproblems. The partitioning can be made by natural hierarchy of decisions (e.g., system design versus system operation), by major resources (e.g., different machines in a production scheduling context), or by chronological time of decisions.

Once the subproblems are defined there are three general solution approaches:

Solve the subproblems independently and somehow coalesce the independent solutions into a feasible solution of the overall problem. An example of production lot-sizing, involving partitioning of time, is provided by Federgruen and Tzur [26]. The traveling salesman problem (TSP) has been solved in this fashion by Karp [26] who partitions the overall geographic region into small regions, solves a TSP for each, then merges the separate tours into a single overall tour.

Solve the subproblems sequentially, using the results of the first as input to the second, etc. This is commonly done in system design followed by system operation. So-called hierarchical planning (Hax, Meal, Bitran and Tirupati) [26] is done in this fashion – aggregate production planning, followed by family scheduling, followed by individual run sizes. Bodin[26] describes a sequential solution procedure for the vehicle routing problem where the number of vehicles (K) is selected, then the customers are separated into K clusters, then a route is chosen for each cluster. The OPT software (Fry et al and Morton and Pentico) [26], based on the theory of constraints, focuses on the bottleneck resource in a multistage scheduling problem, then makes decisions elsewhere to support the smooth functioning of the bottleneck. Finally, in inventory management (Silver et al) [26] one often sequentially chooses the order quantity of an item, then its safety stock or reorder point.

Solve the subproblems iteratively, ie. not just in a sequential fashion. The (shifting) bottleneck dynamics of Morton and Pentico [26] encompasses this approach. They consider the situation of multiple resources shared by multiple activities (e.g. machines shared by jobs in a job-shop scheduling context) and solve single resource (machine) problems iteratively. A key idea in each single resource problem is the estimated marginal benefit of the resource use for each activity.

3.2.3 Inductive Methods

There are two aspects here. First is the generalization from smaller (or somewhat simpler) versions of the same problem or a closely related (from a mathematical pers
pective) problem. The latter embraces the concept of analogy, which also is an important ingredient of creative problem solving. As an example of generalization, Bilde and Vidal [26] considered the problem of locating a number of plants and warehouses. Properties of

the solutions for the cases of several facilities were used to develop a heuristic for the more general case of quite a few facilities. At the opposite extreme, sometimes it is easy to analyze the case where one or more parameters take on very large values, again providing insight for the more difficult case of intermediate values of the parameters.

3.2.4 Methods that Reduce the Solution Space

Namely, the basic idea is to reduce the solution space, ie. cut back drastically on the number of solutions that are even considered while, hopefully, not seriously affecting the quality of the solution obtained. This can be done by tightening existing constraints or by introducing extra constraints. In some cases there may even be an efficient algorithm for the restricted segment of the solution space (e.g. using an optimization routine that is only valid in the restricted region).

One type of approach is to first obtain the optimal solutions to several numerical instances of the problem under consideration (which, of course, may be very difficult to do!). Then an extreme version (eliminating all subsequent search) is to develop regression relationships that give values of the decision variables as functions of key parameters of the problem. An example is the so-called power approximation Ehrhardt [26] used in inventory management. A very flexible form of non- linear regression is achieved through the use of feed-forward neural networks. Such networks consist of a set of nodes connected by directional arcs (without any feedback). The number of layers and the number of nodes in each of the intermediary layers are parameters that can be adjusted. The input signal at a node is a linearly weighted mix of the output signals from other nodes directly linked to it. (The weights are adjustable parameters). The output signal at a node is typically a highly non- linear function of the input signal, e.g. 0 or 1 depending

upon whether or not the input exceeds a threshold. Considerably further details are available in Ignizio and Burke and Michalewicz and Fogel[26].

A more limited restriction of the search space through observing the optimal solutions of a number of problem instances is so-called feature extraction[26]. Conditions that are observed in all (or a great majority) of the optimal solutions are assumed to hold for any future cases to be investigated, ie. the solution is partially specified. Possibilities include:

1. A 0-1 variable that is always 0 or always 1.
2. Two variables that are highly correlated. (e.g. in a facility layout problem two facilities are located beside each other).
3. A constraint that has lots of slack in all the observed solutions is then ignored.
4. A constraint that is always binding.

3.2.5 Approximation Methods

Here we are specifically concerned with manipulating the mathematical model in some way (or using a solution from a related simpler model). The possibilities have been split into four categories:

Aggregation of parameters - The typical approach here is to replace several variables by a single aggregate variable, solve the much smaller, aggregate model, then somehow disaggregate the solution back into a solution of the original problem. Two illustrative applications in logistics decision making are provided by Evans and Geoffrion[26]. Another possibility is to replace a multistage process by an "equivalent" single stage process. Pentico[27] does this for a production problem involving variable yields. A third form of aggregation is to scale the units of each decision variable, for example, to work in

units of 100 instead of 1. Finally, aggregation is possible in dimensions, as illustrated by Bartholdi and Platzman[26], who transform a two-dimensional combinatorial problem (such as the travelling salesman problem) into a related single dimensional problem.

Modification of the objective function – One possibility is to approximate a non- linear function by a piecewise linear one, which may facilitate the use of a linear programming solution algorithm. Rajagopalan[26] describes a related approach, in the context of a make-to-order versus make-to-stock decision, namely using a tractable lower bound on the objective function. An alternative is to simply assume a simpler objective function (ie. use an evaluation function different from the objective function). This is the basic idea in the Silver-Meal heuristic[26] in selecting replenishment lot sizes under a known, but time-varying, demand pattern. In any of these approaches one must ultimately evaluate the performance using the most accurate representation of the true objective function.

Approximating probability distributions or stochastic processes – One option is to assume that random variables are constants at their mean values. Bitran and Yanasse[26] illustrate this idea in the production scheduling of several items, subject to random demands, on a limited capacity machine. We can also use an analytically convenient distribution, such as the normal, having the same mean and variance as the random variable under consideration. This is widely done for the distribution of demand during the replenishment lead time in inventory control models (Silver et al) [33]. Sometimes continuous variables are not attractive in that they imply an infinite number of possibilities. In such circumstances as finite possibilities discrete approximations can be useful. A related approach is to randomly generate or simply select a relatively small set of representative scenarios. Jönsson and Silver [26] have used random generation and Consigli and Dempster [26] have selected a set of scenarios in the contexts of commonality inventory problems and multi-period, portfolio investments, respectively. Stochastic

processes can also be conveniently approximated. For example, the aggregate effect of a large number of renewal processes has a poisson behavior. This property has been used by Silver [26] in an iterative scheme for developing the values of the control parameters in a coordinated inventory control situation.

Change nature of constraints including relaxation methods – First, we can approximate a non- linear constraint by a linear one. We can also choose to completely ignore some constraints, solve the problem and hopefully find that the solution satisfies the constraints. Alternatively constraints can be weakened, e.g. by using surrogate constraints where several constraints are replaced by a single, linear combination of them. In general, the relaxing of constraints can make it easier to solve the resulting model. Some constraints may be flexible in any event (e.g. a budget constraint need not necessarily be rigid). A common relaxation is to permit continuous values of a discrete variable. This may permit the use of calculus to find approximate extreme points. Such a relaxation is also frequently used in linear programming relaxations of (mixed) integer programming problems. Applications include to vehicle routing[33] and to the broad class of multidimensional knapsack problems[33], where a subset of items, each with a given unit value and unit use of one of more resources, such as weight and volume, are to be placed in a container so as to maximize the overall value of the contents, subject to not violating the resource constraints. Relaxation produces a solution, which for a minimization problem gives a lower bound on the objective function value of the optimal solution. Then progressively, heuristically generated constraints are added so that the lower bound can be raised. For a constrained version of the travelling salesman problem, we can produce solutions with objective function values within a prescribed percent deviation of the unknown optimal solution. A widely applicable approach is Lagrangian relaxation[33] whereby one or more of the constraints, multiplied by Lagrange multiplies, are incorporated (relaxed) into the objective function. The multiplier, associated with a

constraint, represents the penalty per unit violation of the constraint. When constraints are relaxed (whether by Lagrangian relaxation or some other method), if the resulting solution is not feasible, it must be appropriately adjusted to achieve feasibility e.g. by some form of local search procedure (the topic of a later section). Beasley and Fisher[26], both, provide an overview of the topic including approaches for obtaining values of the multipliers, also supplying references to a number of practical applications.

3.2.6 Constructive Methods

Constructive methods, as the name implies, use the data of the problem to construct a solution, step by step. Typically, no solution is obtained until the procedure is complete (in contrast with improvement methods to be discussed in the next subsection). A special constructive approach is the so-called greedy method[33], where, at each step, the next element of the solution is chosen so as to give the best immediate benefit (highest profit contribution or lowest cost). The greedy approach is very similar to a sequential myopic perspective, the latter discussed in the earlier section on decomposition/partitioning methods. Ignizio[22], for the case of a problem involving only 0-1 variables, describes two greedy approaches. In the add- heuristic we start with all variables set to 0 and then considers each variable, one at a time. If adding it improves the value of the objective function, then it is set to 1. In the "minus" image, we start with all variables set to 1 (which almost certainly is an infeasible solution). Each variable is considered for deletion and the one doing the least damage to the value of the objective function is set to 0. This is continued until a feasible solution is obtained. Probably the best known greedy application is the nearest neighbour method for the travelling salesman problem (Golden et al)[26]. Specifically, we start at any city (node) and chooses the closest city as the next one to visit, etc. Unfortunately, although extremely easy to use, the greedy approach can lead to a very poor solution, in that the attractive initial choices may result in a very poor

selection near the end. As a result, constructive methods sometimes include some form of look-ahead feature where we estimate the future consequences of the current choice. A more sophisticated, metaheuristic, involving multiple constructive solutions will be covered in a later section.

3.2.7 Local Improvement (Neighbourhood Search) Methods

The basic concept of local improvement methods is quite simple. One starts with a feasible solution to the problem, often the result of a constructive method. Feasible solutions in the neighbourhood [$N(x_c)$] of the current solution (x_c) are evaluated. If one of these is better than the current solution, it becomes the new x_c , its neighbourhood is investigated, etc. until no improvement can be found and the current solution, at that stage, is a local optimum. Not surprisingly for maximization problems neighbourhood search is sometimes called hill climbing. One obvious question is how does one define the neighbourhood of a point (or solution). Müller-Mehrbach[28] has considerable discussion related to neighbourhoods. The neighbourhood $N(x,t)$ is the set of solutions that can be obtained from x by some simple transformation t, ie. different transformations produce different neighbourhoods. Examples include:

In a problem of sequencing a set of jobs on a machine a solution is given by a specific sequence of the jobs and a transformation might be to exchange the order of any two consecutive jobs.

In a model involving 0-1 variables a simple exchange heuristic involves changing one variable's value from 0 to 1 and another's from 1 to 0.

In the travelling salesman problem (TSP) a current solution is a single tour through all cities finishing back in the starting city. A common transformation is to interchange two cities of the tour. For example if a current tour through five cities, A to F, is BDAFC, then switching C and D would lead to a neighbourhood solution BCAFD. More sophisticated transformations are certainly possible. For example, Lin and Kernighan[26] permit up to k edges to be selected for replacement in the TSP.

Another issue is whether to choose a move to the first point in the neighbourhood exhibiting an improvement or to exhaustively evaluate all points in the neighbourhood and choose the one giving the largest improvement. The latter is often referred to as steepest ascent (or descent). Neighbourhood search can also be carried out with continuous variables. The neighbourhood of x might be defined as all points within a certain Euclidean distance of x. If the gradient (partial derivatives of the objective function with respect to each decision variable) can be computed or estimated (which is unlikely) then the concept of steepest ascent (descent) indicates the direction of the move. A control parameter of the heuristic is how far to move in that direction. When the gradient cannot be easily estimated, an alternative is to deterministically specify or randomly generate a subset of the points in the neighbourhood. Random generation could be by adding a normal variable $(0, \sigma_i)$ to the current value of each x_i of the vector solution x. The σ_i is controllable parameter of the heuristic search (see Michalewicz and Fogel[26]). Unfortunately there is a fundamental weakness in local search methods. Only a local optimum is guaranteed, hence, the solution obtained (B) is very much dependent on the starting point (A) and may be quite inferior to the global optimum (C). Not only do we not know if the solution obtained is the global optimum, but even worse we have no idea of how much better the global optimum might be. Local (or neighbourhood) search is very focused. It has been referred to as exploitation or intensification. To break out of the clutches of a local optimum we need a broader search, ie. exploration or diversification,

in other parts of the search space. Of course, one approach would be to significantly increase the size of $N(x_c)$. Unfortunately, the required computational effort quickly explodes as the size increases. Another possibility is to restart the search from a number of points, randomly chosen from the search space. Exploration or diversification is a major ingredient of most metaheuristics, the topic of the next section.

3.3 METAHEURISTICS

A metaheuristic is a higher level heuristic procedure designed to guide other methods or processes towards achieving reasonable solutions to difficult combinatorial optimization problems. Metaheuristics are particularly concerned with not getting trapped at a local optimum (for problems that have multiple local optima) and/or judiciously reducing the search space. Each metaheuristic has one or more adjustable parameters. This permits flexibility, but any application to a specific class of problems requires careful calibration on a set of numerical instances of the problem as well as testing on an independent set of instances. Several metaheuristics are amenable to parallel processing, ie. investigation of different solution sequences can be done in parallel. A total of five different metahe uristics will now be discussed.

3.3.1 Beam Search

The solution space of many combinatorial problems can, in principle, be represented in a tree structure. To be specific, consider two types of problems: i) sequencing problems (eg. The travelling salesman problem), ii) problems where each of a number of variables can take on several discrete values. Suppose that there are n items to be sequenced or n variables involved. Each path from the start node down through n levels represents a

solution. As mentioned earlier, for realistic size problems it is not computationally possible to evaluate all of the paths (solutions). Computation time can be substantially reduced by a branch-and-bound procedure. Suppose that we are dealing with a minimization problem and somehow (e.g. by use of a heuristic!) have obtained a current best complete solution x^* with objective function value $f(x^*)$. Suppose that we are moving from top to bottom of the tree now at node A. The contribution to the objective function of the upper part of the path is easily computable. Let us denote it by CA. If we can obtain a lower bound LBA on any path from A to the bottom of the tree and CA + LBA > $f(x^*)$, then there is no need to consider any of the solutions that include node A. Despite this pruning feature, branch-and-bound still can not guarantee finding the optimal solution to a realistic sized problem.

Beam search (Morton and Pentico)[26,33] is a form of partial branch-and-bound. The basic idea is to discard portions of the tree that are likely, as opposed to guaranteed, to not include the optimal solution. A parameter, called the beam width (w), is the number of nodes that are retained at each level as we proceed down the tree. To rank the nodes for cutting purposes at any level we need a way of estimating the contribution of the best path from each node to the bottom of the tree. To get an accurate estimate may still be very time consuming. A variation, called filtered beam search (Ow and Morton)[26], is to very quickly develop a crude estimate for each node at the level under consideration, then for just the best f (filter width) do a more careful evaluation, and subsequently pick the w (beam width) best of these. Ow and Morton showed that this worked very well on machine scheduling problems. In principle, the beam width need not be the same at all levels of the tree. In particular, if a nominal width of 3 was being used and at a particular level the 3 rd and 4 th best nodes were very close in value, one might profitably change the width to 4 at that level.

3.3.2 Tabu Search

Tabu search is one of the most widely used metaheuristics. Glover[26,37,40] cites some 70 areas of application including vehicle routing, electrical power distribution, transport network design and classroom scheduling. We first describe the basic concepts of tabu search. The method begins with a complete, feasible solution (obtained, e.g., by a constructive heuristic) and, just like local improvement, it continues developing additional complete solutions from a sequence of neighbourhoods. However, to escape from a local optimum, moves to neighbours with inferior solutions are permitted. Moreover, a mechanism is used that prevents cycling back to recently visited solutions (in particular, the local optimum). Specifically, recent solutions (or attributes of solutions) are maintained on a so-called tabu list preventing certain solutions from reoccurring for a certain number of iterations, called the size (or length) of the list. This size is a key controllable parameter of the metaheuristic. A record is maintained of the best solution to date, x^*, and its objective function value $f(x^*)$. The tabu status of a move can be overridden through the use of a so-called aspiration criterion. The simplest version is the following (shown for a maximization problem): if the move leads to a solution x having $f(x) > f(x^*)$, then the move is, of course, permitted. Typically the procedure is terminated after either a prescribed total number of iterations or if no improvement is achieved in some other specified number of consecutive iterations. Just as in basic neighbourhood search a key issue is defining the neighbourhood so that the computational effort is not prohibitive, yet very good solutions are still achieved. The concept of a course filter, discussed under beam search, is relevant here. In its basic form, given the initial solution, tabu search does not include any random elements in contrast with two other metaheuristics to be discussed later, namely simulated annealing and evolutionary algorithms.

There are a wide variety of enhancements of the basic version of tabu search described above. These include:

Dealing with an objective function that is difficult to evaluate – Sometimes it is easier to evaluate the change in the objective function in moving from x_c to a neighbouring solution x. In choosing among neighbours the changes, rather than the absolute values, are sufficient. As in the earlier section on approximate methods one may choose to use an approximate evaluation function, at least for screening purposes. A specific case where it is difficult to evaluate the objective function, is when there are random elements present. Costa and Silver[26] have demonstrated that a form of statistical sampling works quite well in choosing neighbourhood moves.

Dealing with constraints – The most common approach is to use an evaluation function that is the original objective function plus a penalty function for violations of each of the constraints. This is similar in spirit to Lagrangian relaxation, discussed earlier. Costa[26] illustrates this approach in the context of scheduling professional sports matches where constraints include: no more than a certain number of consecutive home games, no more than 3 matches in 4 consecutive days, an even distribution of games throughout the season, etc.

Probabilistic selection of candidate solutions – The usual tabu approach considers the solutions in the neighbourhood $N(x_c)$ of the current solution x_c in a deterministic fashion, possibly according to a priority scheme. Probabilistic tabu search (Glover) [26,33] permits using a random mechanism to choose from a set of candidate solutions where the probabilities can be based on attributes of the solutions.

Variations of the tabu mechanism/list – One possibility is to systematically and/or randomly change the length of the list (Taillard) [26,33]. Another option (Hasegawa et al[26,33]) is to have a continuous (between 0 and 1) tabu status that decays with time. Suppose that at a particular iteration the tabu status of a solution x is t. Then, if x is considered, with probability 1-t its tabu status is ignored. Nanobe and Ibaraki[26] present a method for automatically adaptively adjusting the length of the list based upon performance characteristics in recent iterations.

More sophisticated versions of aspiration criteria - The override of tabu status can be based on specific attributes of the candidate solution, not just its objective function value. One possibility is the degree of "change" in the solution compared with the current one. A large change is desirable if one is attempting to move away from a local optimum.

Frequency – based memory – the usual tabu memory is short term, ie. what has happened recently. In addition, we can maintain a longer-term memory that records frequencies of various solutions (or attributes of solutions). This can be used for two purposes. First, we can intensify, ie. focus, the search in the areas of previously observed, high quality (elite) solutions. Second, in the longer term we can diversify by seeking to generate solutions that are significantly different (as measured by one or more attributes) from those previously encountered. A closely related concept is path relinking (Glover and Laguna) [26,33] where new solutions are generated by starting at a high quality solution and generating paths that are forced to end up at other high quality solutions.

Dealing with continuous variables – Chelouah and Siarry[26] describe the use of tabu search for global optimization when the decision variables are continuous. The neighbourhood of a current solution, x_c , is defined by the region (sphere or hyperrectangle) that is within a certain distance from x_c. Solutions are selected at random

within a series of different sized neighbourhoods. Diversification aspects are also discussed.

There are a number of controllable parameters in a tabu search including the size of the tabu list (and possibly how to dynamically adjust it), how to decide when it is time to diversify, the weights to use in bringing constraints into the evaluation function, etc.

3.3.3 Simulated Annealing

Simulated annealing is another, commonly used, metaheuristic designed to permit escaping from local optima. Applications include the travelling salesman problem, telecommunications network design problem, and facility location decision problem. The name "simulated annealing" is due to the fact that conceptually it is similar to a physical process, known as annealing, where a material is heated into a liquid state then cooled back into a recrystallized solid state. It has some similarities to tabu search. Both start with an initial complete feasible solution and iteratively generate additional solutions, both can exactly or approximately evaluate candidate solutions, both maintain a record of the best solution obtained so far, and both must have a mechanism for termination (a certain total number of iterations or a prescribed number of consecutive iterations without any improvement). However, there are two important differences between the methods. Tabu search permits moving away from a local optimum (ie. diversifying) by an essentially deterministic mechanism, whereas, as we'll see, a probabilistic device is used in simulated annealing. Second, tabu search tends to temporarily permit moving to poorer solutions only when in the vicinity of a local optimum, whereas this can happen at any time in simulated annealing.

At any iteration k we have a current solution x_c and a candidate solution x selected (at random or in a systematic fashion) from the neighbourhood $N(x_c)$. Suppose we are dealing with a maximization problem. As a result, if $f(x) > f(x_c)$, then x becomes the new x_c on the next iteration. If $f(x) < f(x_c)$, then there is still a chance that x replaces x_c, specifically the associated probability is:

$$P(x \to x_c) = \begin{cases} 1 & f(x) > f(x_c) \\ \exp[-\dfrac{f(x_c) - f(x)}{T_k}] & f(x) < f(x_c) \end{cases},$$

where T_k is a parameter called the temperature. The probability of accepting the inferior solution x is seen to decrease as the performance gap between x_c and x increases or as the temperature becomes smaller. The sequence of temperatures usually satisfies $T_1 \geq T_2 \geq \cdots$, ie. the temperature is gradually decreased. There are various mechanisms of achieving this; a common one is to maintain a fixed temperature (T) for a specified number (n) of iterations, then use ΦT for the next n iterations, then $\Phi^2 T$ for the next n iterations etc. where Φ is a controllable parameter satisfying $0 < \Phi < 1$. Decreasing temperatures mean that in the early iterations diversification is more likely, whereas in the later stages the method resembles simple local improvement. Of course, we can use a more sophisticated dynamic control of T where it can, for example, be temporarily increased any time it appears that the procedure has stalled at a local optimum. When the search is terminated it makes sense to do a subsequent local search to ensure that the final solution is at a local optimum.

As with any metaheuristic, there are a number of controllable parameters in simulated annealing, in particular the sequence of temperatures (e.g. the initial temperature and the Φ value) and the termination criterion.

3.3.4 Multi-Start Constructive Approaches – The Adaptive Reasoning Technique

In an earlier section we discussed basic constructive heuristics. Specifically a solution is built up incrementally so that a complete solution is not obtained until the end of the construction. Also mentioned was the so-called greedy approach where at each step the element is added which, considered alone, has the most beneficial impact on the objective function. Multi-start constructive procedures redo the construction many times from different starting points and/or introduce random choice elements. The simplest form is to repeat a constructive heuristic from different starting points. Tsubakitani and Evans[26] have done this for the travelling salesman problem. After generating a number of constructive solutions, they do a regular local search away from the best of these, then away from the second best, etc. The greedy randomized adaptive search procedure (Feo and Resende) [26] starts from a number of random points and at each step of the construction of a solution a random choice is made from a short list of the most greedy elements, ie. the greediest choice is not necessarily made. Again, local improvement of each complete constructive solution is carried out.

The above-discussed multistart procedures do not use information from earlier complete solutions to modify the choice mechanism. Incorporation of this type of feature is the main idea of the adaptive reasoning technique, ART (Patterson, Rolland and Pirkul) [26,33]. Memory is used to aid in learning about the behaviour of a greedy heuristic on a specific problem instance. Constraints are imposed to prevent the greedy heuristic from making, what have been observed to be, poor choices (of specific decision variables). The constraints are dropped after a certain number of iterations (similar to entries on a tabu list). A closely related use of adaptive memory in constructive methods is described by Fleurent and Glover[26,33]. In a sense ART searches for theappropriate set of variables to

not include rather than the usual goal of seeking which variables to include. The method has been applied to several problem areas including workforce assignment and the design of telecommunication networks.

3.3.5 Evolutionary Algorithms

Evolutionary algorithms, as the name implies, are a class of metaheuristics that emulate natural evolutionary processes. Sometimes the adjective "genetic" is used in lieu of "evolutionary". A major portion of the Michalewicz and Fogel book[26] is devoted to the subject. Another general reference is Reeves' book[33] in which several applications (with associated references) are discussed, including the travelling salesman problem.

Evolutionary algorithms work with a group or population of solutions (in marked contrast with earlier discussed metaheuristics). At each iteration each solution in the current population is evaluated. The evaluations serve to select a subset of the solutions to be either used directly in the next population or indirectly through some form of transformation or variation (adjustment of the single solution or generation of a new solution by combining part of the solution with part of another one). The parts of a solution can be thought of as genes, adjustments of single solutions as mutations, and combination as mating to produce offspring. As with the earlier discussed metaheuristics, a record is maintained of the best solution to date. Other issues are representation (how to represent a solution in the form of a vector of genes), choosing the initial solutions, and termination. We next comment briefly on each of these as well as providing some further detail on each of evaluation, selection, and variation.

In a problem with n 0-1 variables a natural representation is simply a vector of n 0-1 genes. In a travelling salesman problem with n cities numbered 1 to n an appropriate

representation is an ordering of the n cities (e.g. 5,3,4,1,2 would represent a tour going from city 5 to 3 to…to 2 to 5). Other types of representations are discussed by Michalewicz and Fogel[26].

The initial population can be simply randomly generated, but it makes sense to include at least one solution that is obtained by a good (constructive) heuristic. Another possibility is to ensure that the initial population is well distributed throughout the solution space. As with the previous metaheuristics, termination results from completing a prespecified number of iterations or through seeing a prespecified (different) number of iterations without improvement. We might think that the evaluation of a solution, sometimes called its fitness, should simply be based on the associated value of the objective function. However, to avoid possible convergence to a population of very similar solutions it may be more appropriate to base the subsequent selection step on a linear transformation of the objective function values or to simply use a ranking, as opposed to absolute continuous values (Reeves) [33].

There are a variety of possible options for the selection step. Some individuals (solutions) can be eliminated from consideration in a deterministic fashion based on their fitness levels.

Alternatively, individuals can be randomly selected (including more than once) for use in the variation phase, where the probability of selection is based on the fitness values (or their rankings).

Selected individuals (solutions) are subjected to forms of variation to produce individuals for the next generation. The most common way of combining two solutions to form two new ones is called simple crossover. The genes of the two parents to the left of the

crossover point are interchanged. More elaborate crossovers are possible using more than one crossover point.

The second common type of variation is mutation where one or more genes of a solution are individually changed with a small probability. There are other forms of variation, partly depending upon the method of representation of a solution (Reeves).

There are a considerable number of controllable parameters and other choices in the use of an evolutionary algorithm to solve a given problem. The parameters include the size of the population, the probability of mutation of an individual gene (which may be best varied during the evolutionary process), the mechanism for generating the size of the mutation change when genes are not just 0-1, the number of crossover points, etc.

There are a variety of enhancements of evolutionary algorithms. Most parallel the earlier discussion related to tabu search. Possible enhancements include the handling of constraints (Michalewicz and Fogel)[26] and continuous variables (Chelouah and Siarry) [26,33], approximate (rather than exact) evaluation of a solution[40], coping with stochastic elements, etc. Finally Glover[26,33,40] describes a more general process, called scatter search[40], for using combinations of several solutions to produce new solutions, (see also Laguna).

3.4 INTERACTIVE METHODS

Interactive (computer/human) procedures have been found to be particularly useful in the development of the model, particularly where the objective function and/or some of the constraints are difficult to explicitly specify. Graphical portrayal of the results of a

tentatively specified model leads to adjustments, etc. (see Bell and Bright and Johnston for illustrative discussions of so-called visual interactive modelling) [29]. In addition, interactive methods can afford advantages in the development and use of heuristic solution procedures, viz.

The graphical representation of a problem, together with a user-friendly interface, can permit the analyst or decision maker to suggest promising solutions, which can then be evaluated by the computer. Fisher[40] describes applications in vehicle scheduling, location decision and production scheduling. Also see Segal and Weinberger[40] for an application concerned with dividing a geographic area into a number of separate customer service areas.

The graphical representation of results (as a function of the number of iterations and the values of other controllable parameters) can facilitate the fine-tuning of metaheuristics.

3.5 EVALUATING THE PERFORMANCE OF A HEURISTIC

There are two broad measures of performance, namely i) how the objective function value obtained compares to that achievable by the optimal solution or some other benchmark procedure, and ii) the computational requirements of the heuristic. With respect to the latter the heuristic should require reasonable computational effort and memory to obtain the solution for realistic sized problems.

The subsequent discussion here will focus on the objective function value achieved. Specifically, it is desirable to have very good average performance, ie. the solution value obtained is close to the optimum value, on average. In addition, robustness is desired in two senses. First, there should be a very low chance of achieving a poor solution. Second,

the performance should not be sensitive to the actual or estimated values of the parameters of the problem. If the results are found to be sensitive, then it is helpful to specify under which conditions the heuristic should be used/not used.

If possible, a histogram of penalties in the objective function value should be obtained. In particular, for what percentage of the problem instances does the heuristic obtain the optimal solution? Also, normally one expresses penalties in a percentage form, viz. (for a minimization problem)

$$\text{Percent penalty} = \frac{f(x_h) - f(x_0)}{f(x_0)} \cdot 100\%$$

where x_h is the heuristic solution and x_0 is the optimal solution.

However, if $f(x_0)$ is very close to or even equal to zero, extremely high percentage penalties will result even if $f(x_h)$ is only slightly above $f(x_0)$. Thus, it is recommended[40] the use of

$$\text{Modified percent penalty} = \frac{f(x_h) - f(x_0)}{f(x_r) - f(x_0)} \cdot 100\%$$

where x_r is some reference solution, such as that obtained using the current decision rule. Also, in some instances, the decision maker might be more interested in the

$$\text{Absolute penalty} = f(x_h) - f(x_0).$$

3.5.1 Experimental Set of Problem Instances

In general, the performance of a heuristic depends upon many parameters (or factors) of the problem. As a result it makes sense to carry out a carefully designed experimental set of test problems, whose results are then statistically analyzed. Typically there are too many factors to permit a complete factorial design of experiments. Insight (partly from an understanding of the related theory), as well as an investigation of the results of preliminary experiments can suggest which variables are likely to be important. The tested values of the factors should be representative of those observed in the real world problem context being studied.

There are two other approaches that have sometimes been advocated in lieu of a designed set of experiments, but both are of limited, practical value. One, so-called probabilistic analysis[40], assumes that each of the parameters follows a prescribed, independent probability distribution and then analytically (typically only possible under very questionable assumptions) or by random sampling develop a probability distribution of the performance of the heuristic. The other, worst case analysis[40], determines the worst possible performance of the heuristic (and the associated values of the problem parameters). However, even when worst case performance is very poor, typical or average performance can often be very good.

3.5.2 Comparison with Optimal Solution

Consider a minimization problem. For any instance of the problem we can plot values of the objective function. Ideally we would like to compare the heuristic solution value with that of the optimal solution. However, as discussed earlier, one of the primary reasons for using heuristics is that it is not practical to find the optimal solution for realistic sized

problems. Thus, comparison may only be possible with smaller scale problems or special cases of larger problems. As a result, it is necessary to do other types of comparisons as discussed in the next two subsections. However, there is another approach for estimating the optimal value, namely so-called extreme value estimation. Suppose that $f(x_1), f(x_2), \cdots, f(x_m)$ represent the values of the solutions of *m* independent trials of a heuristic (involving probabilistic elements) on the same problem instance. Then, based on the assumption that these are independent draws from an extreme value distribution (three-parameter Weibull distribution), one of whose parameters is the (unknown) minimum value, either point or interval estimates of the minimum can be constructed for comparison with the lowest of the *m* values found by the heuristic.

3.5.3 Comparison with Bounds

We continue focussing on a minimization problem. As discussed in the earlier subsection "Approximative Methods" we can obtain a lower bound on the optimal solution value by solving (a typically simpler) problem resulting from relaxing one or less constraints. (Typical relaxations were discussed in that subsection). A different approach to bounding is demonstrated by Klein and Scholl[26,33,40]. For a minimization problem they select a value V of the objective function and, using this as a constraint ($\leqslant$V), they attempt to find at least one feasible solution satisfying this and all the original constraints of the problem. If it can be proved that there is no such feasible solution, which may be a very difficult undertaking as the above modified formulation is the so-called constraint satisfaction problem[26,33,40], which is known to be NP-complete, then V is a valid lower bound for the original problem. We keeps increasing V incrementally until a feasible solution is first found.

If the gap between $f(x_h)$ (objective function value of heuristic solution) and LB (lower bound) is small, then we know that the distance between $f(x_h)$ and the unknown $f(x_0)$ (objective function value of heuristic solution) must be small. However, a large gap between $f(x_h)$ and LB leaves us uncertain as to the performance of the heuristic, in that it can be due to one or both of i). LB is a very weak bound (ie. there is a large gap between $f(x_0)$ and LB) and /or ii). the heuristic has performed poorly (ie. there is a large difference between $f(x_h)$ and $f(x_0)$).

3.5.4 Other Comparisons

Other comparisons are necessary when neither the optimal solution nor a good bound can be obtained. However, there is an even more compelling argument for comparing the performance of the heuristic with that of another method, namely that which has been previously used by the organization under study to make the associated decision(s). "Management is more likely to be convinced of the utility of a heuristic method if it is shown to significantly outperform current practice than by arguments that it comes close to (the relatively vague concept of) optimality"[24]. The heuristic can also be compared with an earlier proposed heuristic for the same problem, but the danger here is that the latter's performance could be quite poor.

Summary

With this chapter of overview of heuristic solution methods, we have pretty much impression regarding how to attack intractable NP-complete problems. As we had referred before, categories of heuristics are not necessarily mutually exclusive. It makes sense to blend more than one type of heuristic, or combine heuristics with traditional

good algorithms in solving a specific class of problems. In the following chapter, we will construct a comprehensive heuristic based on this idea. In this sense, heuristics are more like methods than algorithms. Also, in evaluating the performance of the comprehensive heuristic, we use bound for comparison.

CHAPTER 4

A HEURISTIC METHOD FOR A ROSTERING PROBLEM WITH THE OBJECTTIVE OF EQUAL ACCUMULATED FLYING TIME

INTRODUCTION

In this chapter, we come back to the end of first chapter, namely, a model of huge nonlinear integer programming. The constructive heuristic algorithm we build includes three basic parts: 1, To decrease the scale of the problem, we decompose the whole problem into a series of one-day-phase sub-problems (this is time sequential decomposition); 2,in every phase day, formulate the selection and combination of pilots as a weighted matching problem, and employ polynomial algorithm to solve the weighted matching problem (here, we employ exact algorithm for the sub-problems); 3, after finding the pilots combination for each task in this phase, employ sort-then-assign method and main-pilot-preferred principle to finish the assigning process of this phase(here, we use two heuristic principles).

4.1 ANALYSIS THE PROPERTIES OF THE MODEL

As we had referred in first chapter, at first glimpse, model (1.1-1.12) is a highly complicated nonlinear combinatorial optimization problem. and in detail:

The objective function (1.1) is quadratic. It is a measure of distance from (s_1, s_2) to (s_1*, s_2*), where s_1* is the optimal solution of (1.2-1.6) minimizing (1.2) (we call this problem (a)) and s_2* is the optimal solution of (1.7-1.11) minimizing (1.7) (we call this problem (b)). If there is no constraints group (1.12), we only have two independent sub-models (a) and (b) as follows, which are much easier to deal with:

$$\min s_1^2 = \frac{1}{m_1}\sum_{i=1}^{m_1}(z_i - \frac{1}{m_1}\sum_{i=1}^{m_1} z_i)^2, s_1 \geq 0$$

$$\left\{\begin{array}{ll} z_i = z_i^{(0)} + \sum_{k=1}^{n} x_{ik} T_k & i = 1,2,\cdots,m_1 \\ B_k \geq x_{ik} \cdot E_i^T & i = 1,2,\cdots,m_1, k = 1,2,\cdots,n; \\ \sum_{i=1}^{m_1} x_{ik} = 1 & k = 1,2,\cdots,n; \\ (x_{iq} + x_{ik} - 1) \cdot E_k \leq B_q & i = 1,2,\cdots,m_1, \\ & k,q = 1,2,\cdots,n, \quad k < q; \end{array}\right\} \quad \text{(a)}$$

$$
\left.
\begin{aligned}
&\min s_2^2 = \frac{1}{m_2}\sum_{j=1}^{m_2}(f_j - \frac{1}{m_2}\sum_{j=1}^{m_2} f_j)^2 \,, s_2 \geq 0 \\
&\begin{cases}
f_j = f_j^{(0)} + \sum_{k=1}^{n} y_{jk} T_k & j = 1,2,\cdots,m_2; \\
B_k \geq y_{jk} \cdot E_j^t & j = 1,2,\cdots,m_2, k = 1,2,\cdots,n; \\
\sum_{j=1}^{m_2} y_{jk} = 1 & k = 1,2,\cdots,n; \\
(y_{jq} + y_{jk} - 1) \cdot E_k \leq B_q & j = 1,2,\cdots,m_2; \\
& k,q = 1,2,\cdots,n, \quad k < q;
\end{cases}
\end{aligned}
\right\} \quad \text{(b)}
$$

Problem (a) has bigger feasible solution space for x_{ik} *(i=1,2,⋯,m_1; k=1,2,⋯,n),* so we have $\min_{for\ (1.1\text{–}1.12)} s_1 \geq s_1^* = \min_{for\ (a)} s_1$; Similarly, we have $\min_{for\ (1.1\text{–}1.12)} s_2 \geq s_2^* = \min_{for\ (b)} s_2$; What we want to do is to "drag" (s_1, s_2) toward (s_1*, s_2*), as close as possible.

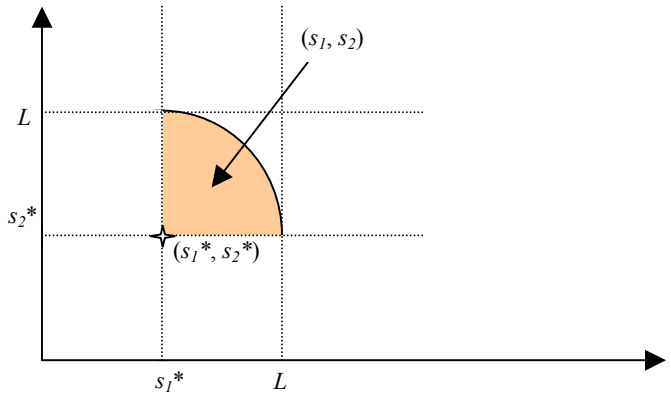

Figure 4.1.1- "drag" (s_1, s_2) toward (s_1*,

Dealing with model (1.1-1.12) means we need at first find s_1* and s_2*, which are the optimal objective values of (a) and (b) respectively. But this is not an easy thing to do since we can prove that (a) and (b) are both NP-hard problems.

The NP-hardness of problems (a), (b) and (1.1-1.12).

Proof: if we make special assumptions to simplify the models, we can get set-partition problem, which is a classical NP-Complete (NPC) problem: Assume that we only have two pilots, which are all main pilots, assume the initial accumulated flying time of this two pilots are zeros, also, assume that: $E_1<B_2$, $E_2<B_3$, $E_3<B_4$, $\cdots$, $E_{n-1}<B_n$, and $E^T_i< B_1$, $i=1,2$.

So, we can reduce the problem (a) and problem (1.1-1.12) to the following one (4.1-4.2):

$$\min \left| \sum_{k=1}^{n} x_{1k} T_k - \sum_{k=1}^{n} x_{2k} T_k \right| \tag{4.1}$$

$$\text{S.T} \quad \begin{cases} x_{1k} + x_{2k} = 1 & k = 1,2,\cdots,n; \\ x_{1k}, x_{2k} = 0 \ \ or \ \ 1, & k = 1,2,\cdots,n; \\ T_k (k = 1,2,\cdots,n) \ \ \text{is positive integer} \end{cases} \tag{4.2}$$

Because the set partition problem (4.1-4.2) is a special case of the rostering problems (a) and (1.1-1.12), and because it belongs to NPC, we have that the rostering problems (a) and (1.1-1.12) are both NP-hard. Similarly, we can argue that rostering problem (b) is also NP-hard.

Because of the NP-hardness, we do not find s_1* and s_2*. We only "drag" s_1 and s_2 below a satisfactory bound. This is the way we try to find satisfactory solution, as shown in Figure 4.1.2. So, what we need to do turn out to be finding an upper bound L. This is the essense of our algorithm in this thesis.

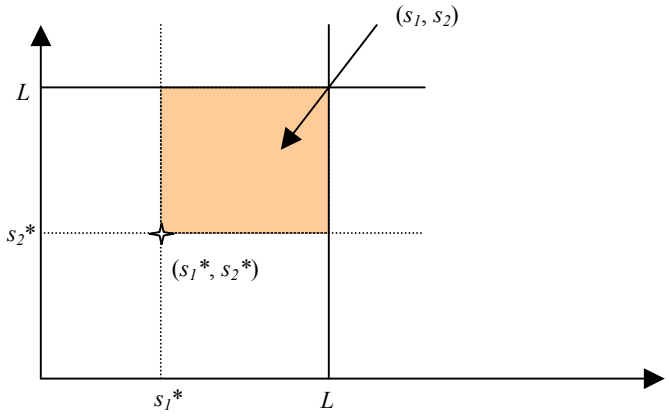

Figure 4.1.2- "drag" s_1, s_2 below a bound L

The scale of the model: the number of the variables is $(m_1 + m_2)n$, the number of constraints is: $m_1 + m_2 + (m_1 + m_2)n + 2n + (m_1 + m_2) \cdot \frac{1}{2} n(n-1) + m_1 m_2 n$. With the increase of the size of m_1, m_2, n, the big number of constraints leads to our giving up most of the methods that dealing with constraints one by one, for example, relaxation technique, evolutionary technique etc. Consider main pilots only, in everage, each main pilot can perform $M=[n/m_1]$ tasks, this leads to more than $\binom{n}{M} \times \binom{n-M}{M} \times \cdots \times \binom{n-[n/M]M}{M}$ options within main pilots group. The big number of options in the case $m_1=58, n= 200$ gives us an impression of the large size of the fessible solution space.

4.2 THE HEURISTIC METHOD FOR FINDING SATISFACTORY SOLUTION

As what we discussed before, we attack problem (1.1-1.12) in decision version: Given an instance (X, Y, s_1, s_2) and a bound L, decide whether there is a feasible solution $x \in X, y \in Y$ with $s_1(x, y) \le L, s_2(x, y) \le L$, where X, Y are the feasible solution space for x_{ik} *(i=1,2,⋯,m_1; k=1,2,⋯,n)* and y_{jk}*(j=1,2,⋯,m_2; k=1,2,⋯,n)*. Theorem 4.2.3.1 gives us a bound $L = \frac{\max_k T_k}{2\sqrt{2}}$, where T_k is the flying time of $Task_k$ *(k=1,2,⋯,n)*. The bound is only determined by the property of the tasks. Because of Airline regulation and agreemnents, maximum value of all tasks makes L a satisfactory bound to evaluate computational results of s_1 and s_2. Theorem 4.2.3.1 also shows that we can control s_1 as $s_1 \le L = \frac{\max_k T_k}{2\sqrt{2}}$ after enough days of assigning within the period considered. We will show how we get this control bound L later on. As for the control of s_2 in (1.7), we will show, numerically, that our algorithm can make it as well as what we do to s_1 in (1.2). If the nonmatched relation between the main pilots and the copilots is sparse, we will see, from the computational results, that in the mean time we can satisfactorily control s_1 in (1.2), we can also satisfactorily control s_2 in (1.7). The assumption of sparse nonmatched relation between the main pilots and the copilots is reasonable in the real operation and management of airline industry. The main-pilot-preferred principle may lead to the worse computational results of s_2 in (1.7) than s_1 in (1.2). However, what we concern is whether or not the controling effects of both are satisfactory. Under the condition of sparse nonmatched relation between the main pilots and the copilots, we can see that our algorithm can output good numerical results. The followings are the details of our heuristic method:

4.2.1 Decomposition

We decompose the whole problem into a series of one-day-phase sub-problems: the following figure shows the sub-problem in the ith phase day.

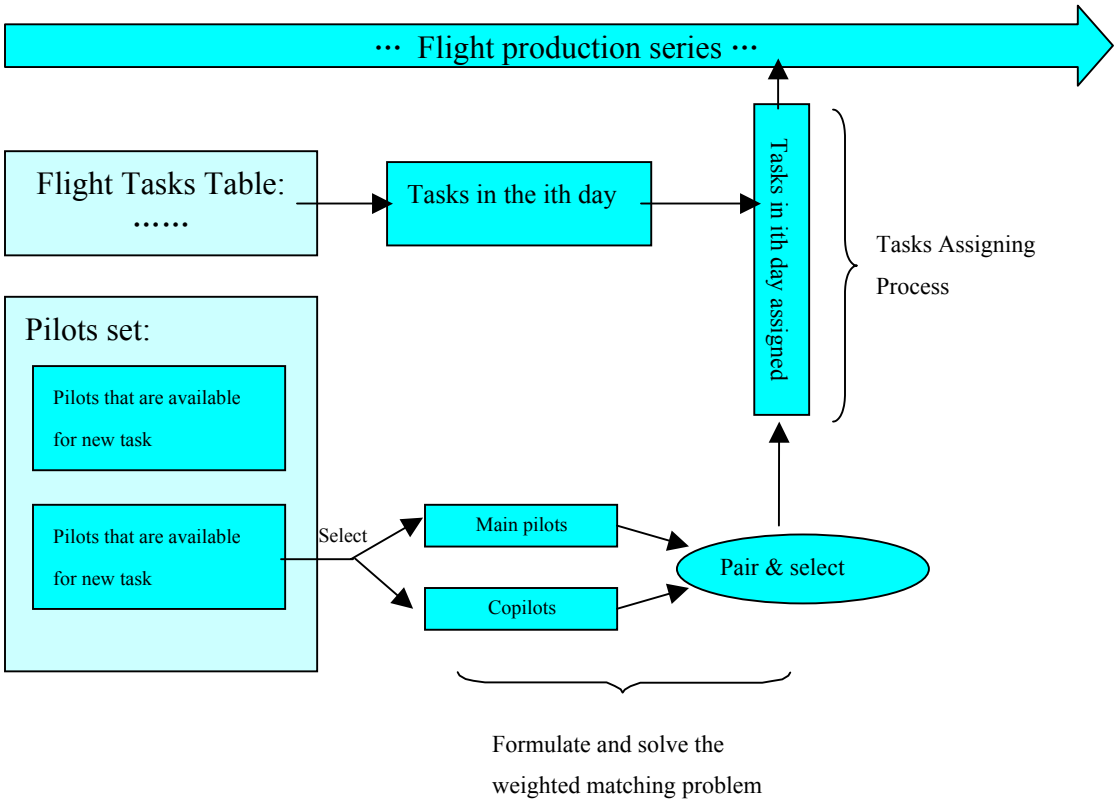

Figure 4.2.1.1-Flights production: assignment

As the Figure 4.2.1.1 above shows, in every phase day we solve two problems: one is the weighted matching problem whose answer will give us the preferred pilots combination

for the tasks of the phase day; the other is how to assign once the pilots combination are found. The following paragraphs will give a detailed introduction to the methods we construct for solving these two problems.

4.2.2 Weighted Matching Problem

If we will assign tasks to pilots on a phase day of the period we consider, first we need to distinguish those who are performing the tasks and those who are waiting for tasks. Once we have done this, we can begin to select from the available pool the pilots that we will dispatch. We do the selection in the way that we both satisfy the requirement of the constraints and control the two standard deviation s_1 and s_2 downward.

Suppose in a phase day of the period that we consider the available pool consists of r_1 main pilots: v_i, $i=1,2,\cdots,r_1$; and r_2 copilots: u_j, $j=1,2,\cdots,r_2$. First, we sort the main pilots and the copilots according to their current accumulated flying time:

$$t(v_1) \le t(v_1) \le \cdots \le t(v_1)$$
$$t(u_1) \le t(u_1) \le \cdots \le t(u_1)$$

Where, $t(\cdot)$ denotes the current accumulated flying time.

If there is no nonmatched relation between any main pilot and any copilot, we prefer a design of pilots combination as:

$$(v_1,u_1),\ (v_2,u_2),\ \cdots\ ,(v_{r1},u_{r1})\quad if\quad r_1 \le r_2$$
$$(v_1,u_1),\ (v_2,u_2),\ \cdots\ ,(v_{r2},u_{r2})\quad if\quad r_1 > r_2$$

where (v_i, u_j) means main pilot v_i and copilot u_j are paired together for a task.

We define this preference as ideal matching without the consideration of nonmatched relation between the main pilots and the copilots. Under the circumstance that nonmatched relation between some main pilots and some copilots exist, we will design a matching as close to the ideal matching as possible. Here we need to formulate the problem as a weighted marching problem.

At first, let $G = (V, U, E, W)$ denote an undirected bipartite graph with $V= \{v_i: i=1,2,\cdots,r_1\}$,$U= \{u_j: j=1,2, \cdots, r_2\}$, $E=\{v_iu_j: i=1,2,\cdots,r_1; j=1,2, \cdots, r_2\}$, and $W=\{w(v_i,u_j) = w_{ij}: i=1,2,\cdots,r_1; j=1,2, \cdots, r_2 \}$, where $w(v_i,u_j) = w_{ij}$ denotes the weight of the edge v_iu_j. Then, we assign value to the weights.

If $r_1 \leqslant r_2$, let:

$$w_{ij} = \left|t(u_i) - t(u_j)\right| + T_{ij}, \quad i = 1,2,\cdots,r_1; j = 1,2,\cdots,r_2$$

If $r_1 > r_2$, let:

$$w_{ij} = \left|t(v_i) - t(v_j)\right| + T_{ij}, \quad i = 1,2,\cdots,r_1; j = 1,2,\cdots,r_2$$

The T_{ij} ($i=1,2,\cdots,r_1; j=1,2, \cdots, r_2$) are the penalty terms we use to express the nonmatched relation between some main pilots and some copilots, we let:

$$\begin{cases} T_{ij} = 0 & \text{if main pilot } v_i \text{ and copilot } u_j \text{ can be paired together for a task;} \\ T_{ij} = +\infty & \text{if main pilot } v_i \text{ and copilot } u_j \text{ can not be paired together for a task} \end{cases}$$

Another way of setting up the value of the weights is to cope with the cases that the nonmatched relation between pilots and copilots may be remarkable:

If $r_1 < r_2$, let:

$$w_{11} = T_{11}, w_{12} = T_{12}, w_{1j} = \left|t(u_1) - t(u_j)\right| + T_{1j}, \quad j = 1,2,\cdots,r_2, j \neq 1,2;$$

$$w_{i,i-1} = T_{i,i-1}, w_{ii} = T_{ii}, w_{i,i+1} = T_{i,i+1}, \quad i = 2,\cdots,r_1,$$

$$w_{ij} = \left|t(u_i) - t(u_j)\right| + T_{ij}, \quad i = 2,\cdots,r_1; j = 1,2,\cdots,r_2, j \neq i-1, i, i+1;$$

If $r_1 = r_2$, let:

$$w_{11} = T_{11}, w_{12} = T_{12}, w_{1j} = \left|t(u_1) - t(u_j)\right| + T_{1j}, \quad j = 1,2,\cdots,r_2, j \neq 1,2;$$

$$w_{i,i-1} = T_{i,i-1}, w_{ii} = T_{ii}, w_{i,i+1} = T_{i,i+1}, \quad i = 2,\cdots,r_1 - 1,$$

$$w_{ij} = \left|t(u_i) - t(u_j)\right| + T_{ij}, \quad i = 2,\cdots,r_1 - 1; j = 1,2,\cdots,r_2, j \neq i-1, i, i+1;$$

$$w_{r_1,r_1-1} = T_{r_1,r_1-1}, w_{r_1,r_1} = T_{r_1,r_1},$$

$$w_{r_1,j} = \left|t(u_{r_1}) - t(u_j)\right| + T_{r_1,j}, \quad j = 1,2,\cdots,r_2, j \neq r_1 - 1, r_1.$$

If $r_1 > r_2$, let:

$$w_{11} = T_{11}, w_{21} = T_{21}, w_{i1} = \left|t(v_1) - t(v_i)\right| + T_{i1}, \quad i = 1,2,\cdots,r_1, i \neq 1,2;$$

$$w_{j-1,j} = T_{j-1,j}, w_{jj} = T_{jj}, w_{j+1,j} = T_{j+1,j}, \quad j = 2,\cdots,r_2,$$

$$w_{ij} = \left|t(v_i) - t(v_j)\right| + T_{ij}, \quad j = 2,\cdots,r_2; i = 1,\cdots,r_1, i \neq j-1, j, j+1;$$

The T_{ij} ($i=1, 2, \cdots, r_1; j=1, 2, \cdots, r_2$) are the penalty terms denoting the nonmatched relation between some main pilots and some copilots, we let:

$$\begin{cases} T_{ij} = 0 & \text{if main pilot } v_i \text{ and copilot } u_j \text{ can be paired together for a task;} \\ T_{ij} = +\infty & \text{if main pilot } v_i \text{ and copilot } u_j \text{ can not be paired together for a task} \end{cases}$$

Of cause, the nonmatched relation between pilots and copilots may be up to the extent that we can not find satisfactory matching as compared with the ideal matching. But as we had pointed before, in practical application, the data supports our assumption of sparse nonmatched relation between pilots and copilots.

After we set up the value of the weights of the bipartite graph G (V, U, E, W), we can propose the model of the weighted matching problem. Once we solve the weighted matching problem, we find the combination of the main pilots and the copilots for the assigning of the phase day we consider.

$$\min \sum_{i=1}^{r_1} \sum_{j=1}^{r_2} \tau_{ij} \cdot w_{ij} \tag{4.3}$$

$$s.t: \begin{cases} \sum_{i=1}^{r_1} \tau_{ij} \le 1, \qquad j = 1,2,\cdots,r_2; & (4.4) \\ \sum_{j=1}^{r_2} \tau_{ij} \le 1, \qquad i = 1,2,\cdots,r_1; & (4.5) \\ \sum_{i=1}^{r_1} \sum_{j=1}^{r_2} \tau_{ij} = R(\text{a given positive integer}) & (4.6) \\ \tau_{ij} = 0 \quad or \quad 1 \quad i = 1,2,\cdots,r_1; j = 1,2,\cdots,r_2 \end{cases}$$

The model (4.3-4.6) is the mathematical description of the weighted matching problem of bipartite graph [31, 41]. The positive integer R is the number of edges of the matching we want to find.

Problem (4.3-4.6) is a special linear integer programming. There are many ways through which we can solve this problem. The most basic algorithm is something like Branch and Bound method, Cutting-Plane method. The better choices are Linear Continuous Relaxation method [31,41], Network Simplex method [31,41], Hungarian method [31,41], Out-of-kilter method [31,41], Cycle Canceling method and Min Cost Flow methods [31,41] etc. In

this thesis we employ Min Cost Flow method, which, in our cases, is polynomial algorithm [41].

The basic idea of Min Cost Flow method is to transform the weighted matching problem of bipartite graph into a min-cost-flow problem [31,41]. In solving the min-cost-flow problem, we use the technique associated with *augmenting path* and *shortest path*. We find *augmenting path* from *the flow digraph*. We find *shortest path* from *the cost digraph.* Once we find an augmenting path from the flow digraph. We can update the flow digraph along the augmenting path to get a new feasible flow that has a greater flow value. The problem of finding an augmenting path from the flow digraph is equal to finding a path from the cost digraph. Therefore, by finding a shortest path from the cost digraph, we can find a *min cost augmenting path* from the flow digraph. If every update in the flow graph happens along a min cost augmenting path, we will finally get a min cost flow with the flow value that we want. Then, back to the weighted matching problem of the bipartite graph, we get the matching we want. Because we need to find the shortest path from the cost digraph, there exists a problem of finding shortest path, we will use Bellman Ford algorithm [31,41] because the cost digraph may contain the arcs of negative weight.

For the undirected bipartite graph $G = (V, U, E, W)$, let's first construct the associated flow digraph $N_1=(s, t, V, U, A_1, C, F)$, where:

s: the starting node;

t: the ending node;

$V= \{v_i: i=1,2,\cdots,r_1\}$: set of nodes denoting the main pilots;

$U= \{u_j: j=1,2, \cdots, r_2\}$: set of nodes denoting the copilots;

$A_1=\{(s, v_i), (v_i, u_j), (u_j, t): i=1,2,\cdots,r_1; j=1,2, \cdots, r_2\}$: set of the arcs;

$C=\{c(s, v_i)=1, c(v_i, u_j)=1, c(u_j, t)=1: i=1,2,\cdots,r_1; j=1,2, \cdots, r_2\}$: set of the flow capacity of the arcs;

$F=\{f(s, v_i)=0$ or $1, f(v_i, u_j)=0$ or $1, f(u_j, t)=0$ or $1: i=1,2,\cdots,r_1; j=1,2, \cdots, r_2\}$: set of the flow along the arcs;

Definition 4.2.2.1: a flow F of the flow digraph N_1 from s to t is *feasible flow* if the conservation condition is satisfied:

$$f(s,v_i)=\sum_{j=1}^{r_2} f(v_i,u_j), \quad i=1,2,\cdots,r_1 ; \tag{4.7}$$

$$f(u_j,t)=\sum_{i=1}^{r_1} f(v_i,u_j), \quad j=1,2,\cdots,r_2 \tag{4.8}$$

Remark: obviously, according to the constraints on the capacities of the arcs we have:

$$f(s,v_i)=\sum_{j=1}^{r_2} f(v_i,u_j)\le 1, \quad i=1,2,\cdots,r_1 ; \tag{4.9}$$

$$f(u_j,t)=\sum_{i=1}^{r_1} f(v_i,u_j)\le 1, j=1,2,\cdots,r_2 \tag{4.10}$$

Definition 4.2.2.2: for a positive integer R not big enough, given a R-flow-value feasible flow F of the flow digraph N_1, its cost is defined as:

$$b(F)=\sum_{i=1}^{r_1}\sum_{j=1}^{r_2} f(v_i,u_j)\cdot w_{ij} \tag{4.11}$$

where w_{ij} denotes the weight of the edge $v_i u_j$ of the undirected bipartite graph $G = (V, U, E, W)$.

Remark: we can add node s and node t to the undirected bipartite graph $G = (V, U, E, W)$ to get an undirected graph $G_1 = (s, t, V, U, E_1, W_1)$, where $E_1=\{E, \{sv_i: i=1,2,\cdots,r_1\}, \{u_jt: j=1,2, \cdots, r_2\}\}$, and $W_1=\{W, \{w(s,v_i)= w_{si} =0, i=1,2,\cdots,r_1\}, \{w(u_j, t)= w_{jt}=0; j=1,2, \cdots, r_2\}\}$, so, the original definition of the flow cost should be:

$$b(F) = \sum_{i=1}^{r_1} f(s, v_i) \cdot w_{si} + \sum_{j=1}^{r_2} f(u_j, t) \cdot w_{jt} + \sum_{i=1}^{r_1} \sum_{j=1}^{r_2} f(v_i, u_j) \cdot w_{ij} \qquad (4.12)$$

However, the first two partial sums vanish so that the definition of the flow cost is simplified.

Theorem 4.2.2.1: there exists a 1-1 mapping from the matching set of the undirected bipartite graph $G = (V, U, E, W)$ onto the feasible flow set of the flow digraph N_1.
Proof: by (4.9-4.10), we say that finding an edge of a matching is equivalent to finding a flow $f(v_i, u_j) = 1$ of a feasible flow, so the conclusion of the theorem follows.

Corollary: given a positive integer R, if R is not big enough, there exists a 1-1 mapping from the min weight R-edges matching set of the undirected bipartite graph $G = (V, U, E, W)$ onto the min cost R- flow-value feasible flow set of the flow digraph N_1.
Proof: the theorem 4.2.2.1 and identity (4.11) show the conclusion.

Remark: this corollary is the foundation that we can transform the weighted matching problem of bipartite graph into a min-cost-flow problem.

Definition 4.2.2.3: for a feasible flow F of the flow digraph $N_1 = (s, t, V, U, A_1, C, F)$ from s to t, an *augmenting path* $\boldsymbol{P}$ is a path from s to t in the undirected graph resulting from N_1 by ignoring arc directions, with the following properties:

For every arc $(\alpha, \beta) \in A_1$ that is traversed by $\boldsymbol{P}$ in the forward direction (called a forward arc), we have $f(\alpha, \beta) = 0$. That is, forward arcs of $\boldsymbol{P}$ are unsaturated.

For every arc $(\alpha, \beta) \in A_1$ that is traversed by $\boldsymbol{P}$ in the reverse direction (called a backward arc), we have $f(\alpha, \beta) = 1$. That is, backward arcs of $\boldsymbol{P}$ are not zero.

Definition 4.2.2.4: given a feasible flow F of the flow digraph N_1, a *flow adjustment* along an augmenting path is defined as:

$$f'(\alpha,\beta) = \begin{cases} f(\alpha,\beta)+1 & (\alpha,\beta) \in P^+ \\ f(\alpha,\beta)-1 & (\alpha,\beta) \in P^- \end{cases} \tag{4.13}$$

where P^+ denotes the set of forward arcs, and P^- denotes the set of backward arcs.

Theorem 4.2.2.2: for a R- flow-value feasible flow F of the flow digraph N_1, after a flow adjustment along an augmenting path $\boldsymbol{P}$, the new flow $F \leftarrow \{\{f'(\alpha, \beta):(\alpha, \beta) \in \boldsymbol{P}\}, \{f(\alpha, \beta): (\alpha, \beta) \notin \boldsymbol{P}\}$ is still a feasible flow and has flow value of $R+1$.

Proof: the flow adjustment formula (4.13) implies that the conservation condition (4.7-4.8) still holds. So, the first conclusion follows. The flow adjustment formula (4.13) also shows that:

$$\sum_{i=1}^{r_1} f'(s,v_i) = 1 + \sum_{i=1}^{r_1} f(s,v_i) = R+1 \tag{4.14}$$

$$\sum_{j=1}^{r_2} f'(u_j,t) = 1 + \sum_{j=1}^{r_2} f(u_j,t) = R+1 \qquad (4.15)$$

which proves the second conclusion.

Remark: this theorem tells that, when finding an augmenting path, we can increase the flow value while we find new feasible flow of the flow digraph N_1. So, starting from a *0*-flow, which is the most basic initial feasible flow with the cost of zero, we can update one and another until we get a feasible flow with the flow value that we want.

The question is: when we do this, how could we minimize the increment of the flow cost in every step? That is the problem of finding min cost augmenting path.

Definition 4.2.2.5: the cost of an augmenting path ***P*** (denoted as $b(P)$) associated with a *R*- flow-value feasible flow F of the flow digraph N_1 is defined as the cost increment, when along ***P***, F is updated into F', the flow value is increased from R up to $R+1$, with the following expression:

$$b(P) = b(F') - b(F) = \sum_{(\alpha,\beta)\in P} w(\alpha,\beta)\cdot f'(\alpha,\beta) - \sum_{(\alpha,\beta)\in P} w(\alpha,\beta)\cdot f(\alpha,\beta) \qquad (4.16)$$

where $w(\alpha, \beta) \in W_1$, which is subset of the undirected graph G_1. We use $(\alpha, \beta) \in \boldsymbol{P}$ to denote that the arc (α, β) is traversed by ***P***. $b(F')$, $b(F)$ denote the cost of flow F' and F, respectively.

Remark: because the arcs along the augmenting path $\boldsymbol{P}$ can be divided into forward arcs and backward arcs, we let $\boldsymbol{P}^+$ denote the set of forward arcs traversed by $\boldsymbol{P}$ and let $\boldsymbol{P}^-$ denote the set of backward arcs traversed by $\boldsymbol{P}$. So, we can rewrite (4.16) as:

$$\begin{aligned} b(P) = b(F') - b(F) = & \sum_{(\alpha,\beta)\in P^+} w(\alpha,\beta)\cdot[f'(\alpha,\beta) - f(\alpha,\beta)] + \\ & \sum_{(\alpha,\beta)\in P^-} w(\alpha,\beta)\cdot[f'(\alpha,\beta) - f(\alpha,\beta)] \\ = & \sum_{(\alpha,\beta)\in P^+} w(\alpha,\beta) - \sum_{(\alpha,\beta)\in P^-} w(\alpha,\beta) \end{aligned} \tag{4.17}$$

So, the problem of finding min cost augmenting path is to find $\boldsymbol{P}$ such that (4.17) can be minimized in every step of updating.

For the undirected bipartite graph $G = (V, U, E, W)$, let's construct the associated cost digraph $N_2=(s, t, V, U, A_2, b)$, where:

s: the starting node;

t: the ending node;

$V= \{v_i: i=1,2,\cdots,r_1\}$: set of nodes denoting the main pilots;

$U= \{u_j: j=1,2,\cdots, r_2\}$: set of nodes denoting the copilots;

$A_2=\{(s, v_i), (v_i, s), (v_i, u_j), (u_j, v_i), (u_j, t), (t, u_j): i=1,2,\cdots,r_1; j=1,2,\cdots, r_2\}$: set of the arcs;

$b=\{b(s, v_i), b(v_i, s), b(v_i, u_j), b(u_j, v_i), b(u_j, t), b(t, u_j): i=1,2,\cdots,r_1; j=1,2,\cdots, r_2\}$: set of the cost of the arcs;

We also define b as associated with a feasible flow F of the flow digraph N_1. For every arc $(\alpha, \beta) \in A_1 \in N_1$, let:

$$b(\alpha,\beta) = \begin{cases} w(\alpha,\beta) & if \quad f(\alpha,\beta) = 0 \\ +\infty & if \quad f(\alpha,\beta) = 1 \end{cases}; \tag{4.18}$$

$$b(\beta,\alpha)=\begin{cases}-w(\alpha,\beta) & if \quad f(\alpha,\beta)=1\\ +\infty & if \quad f(\alpha,\beta)=0\end{cases} \tag{4.19}$$

Theorem 4.2.2.3: the problem of finding a min cost augmenting path from the flow digraph N_1 is equal to finding a shortest path from N_2

Proof: let's look at what is the cost of a path in the cost digraph N_2.From (4.18-4.19), a cost of path ρ in the cost digraph N_2 can be written as:

$$\begin{aligned}&\sum_{(\alpha,\beta)\in\rho} b(\alpha,\beta)+\sum_{(\beta,\alpha)\in\rho} b(\beta,\alpha)\\ &=\sum_{f(\alpha,\beta)=0} w(\alpha,\beta)-\sum_{f(\alpha,\beta)=1} w(\alpha,\beta)\\ &=\sum_{(\alpha,\beta)\in P^+} w(\alpha,\beta)-\sum_{(\alpha,\beta)\in P^-} w(\alpha,\beta)\end{aligned} \tag{4.20}$$

which is the cost of an augmenting path $\boldsymbol{P}$ in the flow digraph N_1. Because the steps in (4.20) is inversable, we can say that for every augmenting path of a cost in the flow digraph N_1, there exists one and only one path of the same cost in the cost digraph N_2.So, the conclusion of the theorem follows.

Remark: this theorem transforms our original weighted matching problem into the problem of finding shortest path in a digraph. The later is much easier to solve. Because there exists negative weights in the arcs, and there is only one starting node *s* and only one ending node *t*. We use the 3rd order polynomial algorithm Bellman-Ford.

In solving the shortest path problem of the cost digraph N_2, for convenience, let's write: $a_1 = s$, $a_2 = v_1$, ⋯, $a_{r1+1} = v_{r1}$, $a_{r1+2} = u_1$, ⋯, $a_{r1+r2+1} = u_{r2}$, $a_{r1+r2+2} = t$. And let $n = r_1+r_2+2$.The Bellman-Ford can be written as a system of iteration equations:

$$L^{(1)}(a_1, a_1) = 0, \tag{4.21}$$

$$L^{(1)}(a_1, a_j) = b(a_1, a_j), j = 2, \cdots, n \tag{4.22}$$

$$L^{(k+1)}(a_1, a_1) = 0, \tag{4.23}$$

$$L^{(k+1)}(a_1, a_j) = \min\{L^{(k)}(a_1, a_j), \min_{i:i \neq j}(L^{(k)}(a_1, a_i) + b(a_i, a_j))\},$$
$$j = 2, \cdots, n; k = 1,2, \cdots, n-2. \tag{4.24}$$

From this system of iteration equations, we can directly get the following theorem:

Theorem 4.2.2.4: in (4.21-4.24) the label $L^{(k)}(a_1, a_j), j \neq 1$ is the length of the shortest path from a_1 to a_j (j=2, 3,⋯,n),with at most k arcs.

Remark: according to (4.21-4.24), if the cost digraph N_2 does not contain *negative cycle*, $L^{(k)}(a_1, a_j), j=2, 3, \cdots, n$ convergence when $k \leq n-1$.

Remark: until now, we have all the staffs we may need to solve the weighted matching problem. The overall complexity of the Min Cost Flow algorithm can be written as: $O(R \cdot S)$, where $S = O(m \cdot n)$ is the complexity of the Bellman-Ford algorithm, n is the number of nodes, m is the number of arcs, R ($\leq \min(r_1, r_2)$) is the flow value that we want, it is a given positive integer. So, the overall complexity is polynomial, the order is no more than 4^{th}.

4.2.3 Sort-Then-Assign Method

In a phase day of the period we consider, by formulating finding pilots combination into weighted matching problem and solving it, we can, as long as the 0-1 parameter matrix $(p_{ij})_{m1 \times m2}$ is sparse enough, pair the main pilots of long accumulated flying time with the

copilots of long accumulated flying time; pair the copilots of short accumulated flying time with the copilots of short accumulated flying time. We also can get enough pairs as long as we have enough pilots.

Main-pilot-preferred principle: after we get the pairs by solving the weighted matching problem, we assign the tasks to pilots. We sort the pilot pairs in ascending order according to the accumulated flying time of main pilots of the pairs. We sort the tasks of this phase day in descending order according to the flying time of the tasks. In assigning, we assign task of long flying time to the pilots pair with short accumulated time of main pilot.

The following Figure gives a clear description to the sort-then-assign method and the main-pilot-preferred principle:

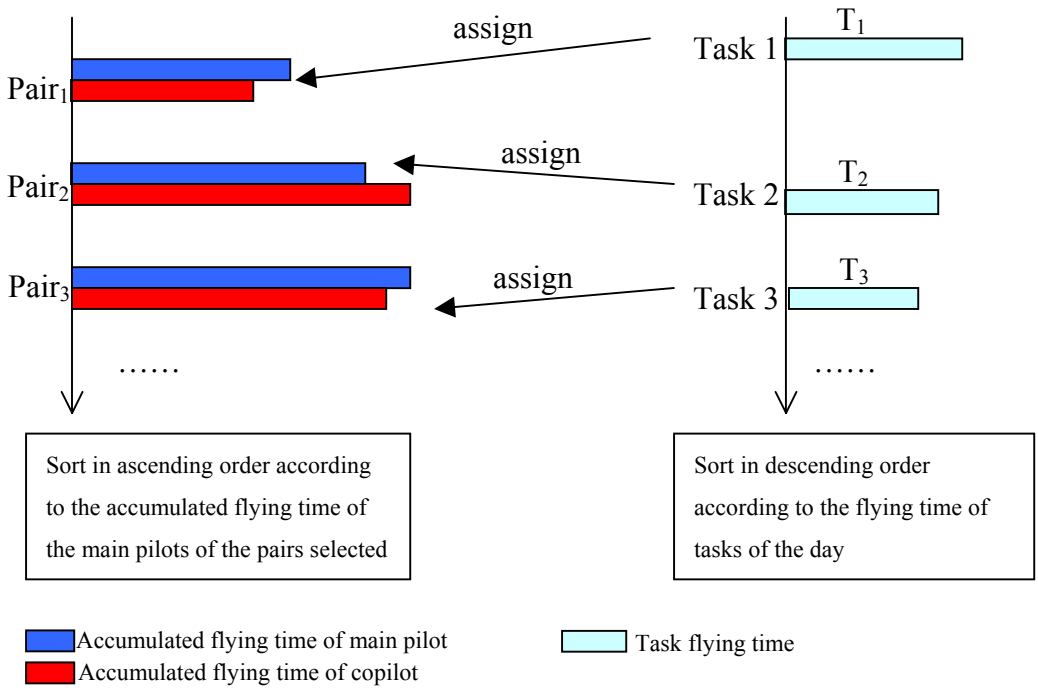

Figure 4.2.3.1- sort-then-assign method and the main-pilot-preferred principle

Theorem 4.2.3.1: If there are enough pilots to ensure accomplishing the tasks of every phase day, after assigning enough tasks, the standard deviation in objective function (1.2) can be bounded as: $s_1 \le \frac{\max_k T_k}{2\sqrt{2}}$, where T_k is the flying time of $TASK_k$, $k=1,2,\cdots,n$.

Proof: Consider the ith main pilot and the jth main pilot, $i, j=1,2,\cdots,m_1,\ i \neq j$. Suppose before a phase day of assigning, the accumulated flying time is counted as z_i and z_j, respectively. After the phase day of assigning, the accumulated flying time is counted as z_i' and z_j', when added T_i (≥ 0) and T_j (≥ 0), respectively. $T_i = 0$ or $T_j = 0$ means no task assigned. Without losing generality, let $z_i \le z_j,\ T_i \ge T_j$. So, we should have:

$$\begin{aligned} \left|\Delta z_{ij}^{'}\right| = \left|z_j^{'} - z_i^{'}\right| &= \left|z_j + T_j - (z_i + T_i)\right| \\ &= \left|z_j - z_i + T_j - T_i)\right| \\ &= \left|\left|\Delta z_{ij}\right| - \left|\Delta T_{ij}\right|\right| \end{aligned} \tag{4.25}$$

So, it is not hard to find that:

$$\left|\Delta z_{ij}^{'}\right| < \left|\Delta z_{ij}\right| \text{ if } \left|\Delta z_{ij}\right| > \frac{1}{2}\left|\Delta T_{ij}\right| > 0; \tag{4.26}$$

$$\left|\Delta z_{ij}^{'}\right| = \left|\Delta z_{ij}\right| \text{ if } \left|\Delta z_{ij}\right| = \frac{1}{2}\left|\Delta T_{ij}\right| > 0 \text{ or } \left|\Delta T_{ij}\right| = 0; \tag{4.27}$$

$$\left|\Delta z_{ij}^{'}\right| > \left|\Delta z_{ij}\right| \text{ if } \left|\Delta z_{ij}\right| < \frac{1}{2}\left|\Delta T_{ij}\right|. \tag{4.28}$$

From (4.26-4.28), we can find that, after assigning enough tasks, any $|\Delta z_{ij}| = |z_i - z_j|$, $i, j=1,2,\cdots,m_1,\ i \neq j$ satisfies:

$$\left|\Delta z_{ij}\right| = \left|z_i - z_j\right| < \frac{1}{2}\max_{ij}\left|\Delta T_{ij}\right| = \frac{1}{2}(\max_k T_k - 0) = \frac{1}{2}\max_k T_k \tag{4.29}$$

Because for $i, j=1,2,\cdots,m_1, i \neq j$, we have:

$$\begin{aligned}
&\sum_{i,j}(z_i - z_j)^2 \\
&= \sum_{i,j}(z_i - \bar{z} + \bar{z} - z_j)^2 \\
&= \sum_{i,j}\left[(z_i - \bar{z})^2 - 2(z_i - \bar{z})(z_j - \bar{z}) + (\bar{z} - z_j)^2\right] \\
&= \sum_{i,j}(z_i - \bar{z})^2 - 2\sum_{i,j}(z_i - \bar{z})(z_j - \bar{z}) + \sum_{i,j}(\bar{z} - z_j)^2 \\
&= 2m_1\sum_{i}(z_i - \bar{z})^2 - 2[\sum_{i}(z_i - \bar{z})]^2 \\
&= 2m_1(m_1 - 1)s_1^2 \leq m_1(m_1 - 1)(\frac{1}{2}\max_k T_k)^2
\end{aligned}$$

So,

$$s_1 \leq \frac{1}{2\sqrt{2}}\max_k T_k \tag{4.30}$$

This tells the conclusion of the theorem.

Remark: reviewing all the stuffs we introduce to solve the rostering problem, we can find that the overall complexity of the heuristic algorithm consists of the complexity of two sorting phases and a phase of weighted matching problem. We have known that the complexity of Min Cost Flow algorithm is no more than 4th order. We also know that the complexity of the algorithm for sorting can be no more than 2nd order. So, the overall complexity of our heuristic algorithm is no more than 4th order: O ($R \cdot S$) + O ($m_1^2 + m_2^2$) + O (R^2) = O ($R \cdot$ O ($m \cdot n$)) + O ($m_1^2 + m_2^2$) + O (R^2).

Remark: as what we had introduced before, as long as there are enough pilots, we can satisfactorily bound s_1, the standard deviation in objective function (1.2). Then, the question regarding how about the control of s_2, the standard deviation in objective function (1.7), follows. Obviously, when the 0-1 parameter matrix $(p_{ij})_{m1\times m2}$ is not a zero one, the main-pilot-preferred principle may lead to the bigger result of s_2 than s_1, or even lead to a bad result of s_2.

However, several other factors also play roles in affecting the control of s_2. These factors are:

1. The initial distribution of accumulated flying time.
2. The initial state of tasks performing.
3. The number of the pilots.

Of course, these factors also have effects on the control of s_1. What we want to know is: with enough pilots, under the circumstance of sparse nonmatched relation between the main pilots and the copilots, can we satisfactorily bound s_1 and, at the same time, s_2? Is the time it takes us to compute satisfactory or acceptable? What influence does the main-pilots-preferred principle have? How do the three factors (1-3) affect the control of s_1 and s_2? In the following numerical tests of our heuristic method, we will carry out the corresponding research.

4.3 IMPLEMENTATION SCHEME OF THE HEURISTIC METHOD FOR FINDING SATISFACTORY SOLUTION

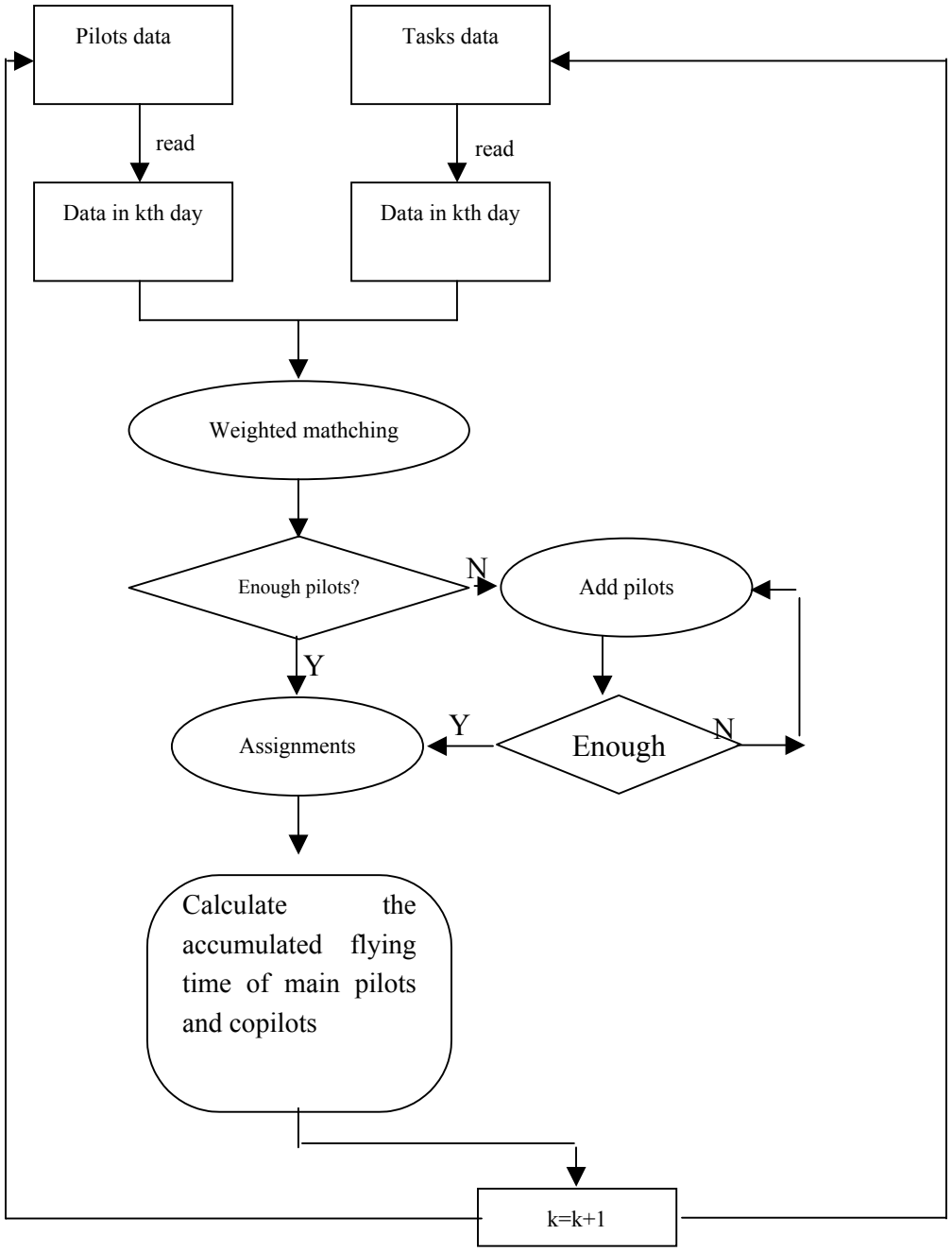

Figure 4.3.1-Flow chart of the algorithm implementation

4.4 NUMERICAL TESTS OF THE HEURISTIC METHOD

Here we use a monthly flight tasks table that contains 17 tasks each day. The maximum flying time of a task is 805 minites, which means our control bound in (4.30) is $\frac{1}{2\sqrt{2}}\max_k T_k = 284.61 \text{minutes}.$

Table 4.4.1 shows the nonmatched relation between 58 main pilots and 62 copilots. Table 4.4.2 shows the distribution of accumulated flying time of main pilots and copilots after 30 days' assigning, given an initial condition of initial tasks performing state plus a distribution of initial accumulated flying time.

Figure 4.4.1 shows the initial condition. From the initial tasks performing state we can tell that there is not so much pilots redundancy that we must wait for some pilots to finish their current tasks to take new tasks. In Figure 4.4.1, "1" denotes state of performing task; "0" denotes waiting task. Other figures have the same denotation. Figure 4.4.2 shows that both the s_1 and the s_2 are controlled very well, given the value (284.1) of control bound (4.30).

Table 4.4.1: Nomathced relation1 (m_1=58, m_2=62)

ID No. of the main pilots	ID No. of the nonmatched copilots
3	59, 65, 73
9	59, 71, 77, 80
33	72, 75, 87
41	67
52	72, 75

Table 4.4.2:Computational result 1 (m_1=58, m_2=62)

The number of pilots		Main pilots: 58	Copilots: 62
At the beginning	The standard deviation of accumulated flying time (minutes)	741.16	742.00
The maximum standard deviation of Accumulated flying time (minutes)		741.16	727.89
In the end (30^{th} day)	The standard deviation of accumulated flying time (minutes)	178.18	178.39
CPU (Intel Celeron 900) time (in minutes)		3.17	
The control bound: $\frac{\max_k T_k}{2\sqrt{2}}$		284.61	

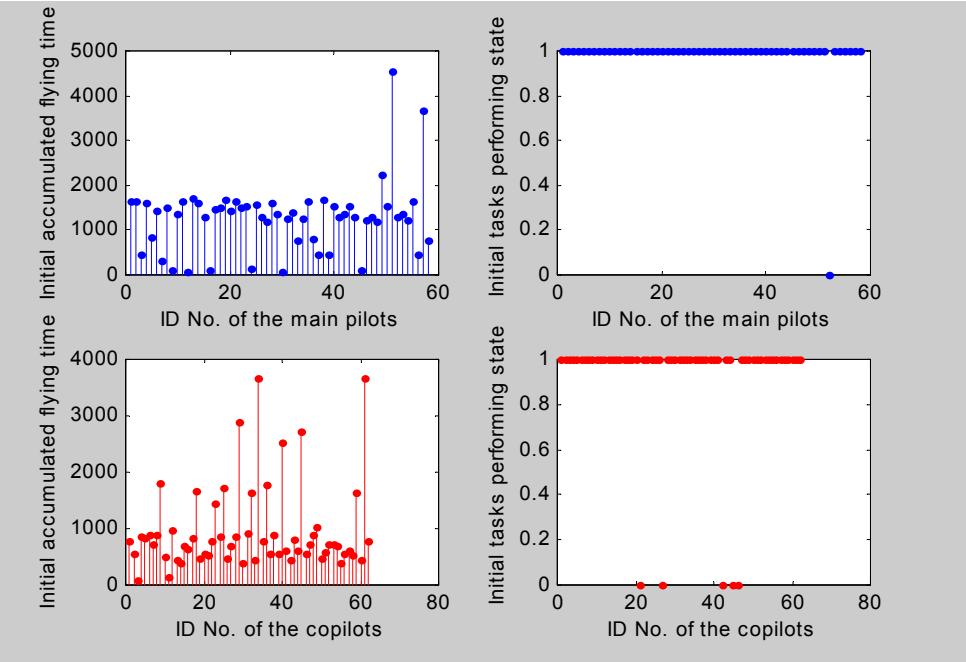

Figure 4.4.1 Innitial condition 1 (m_1=58, m_2=62)

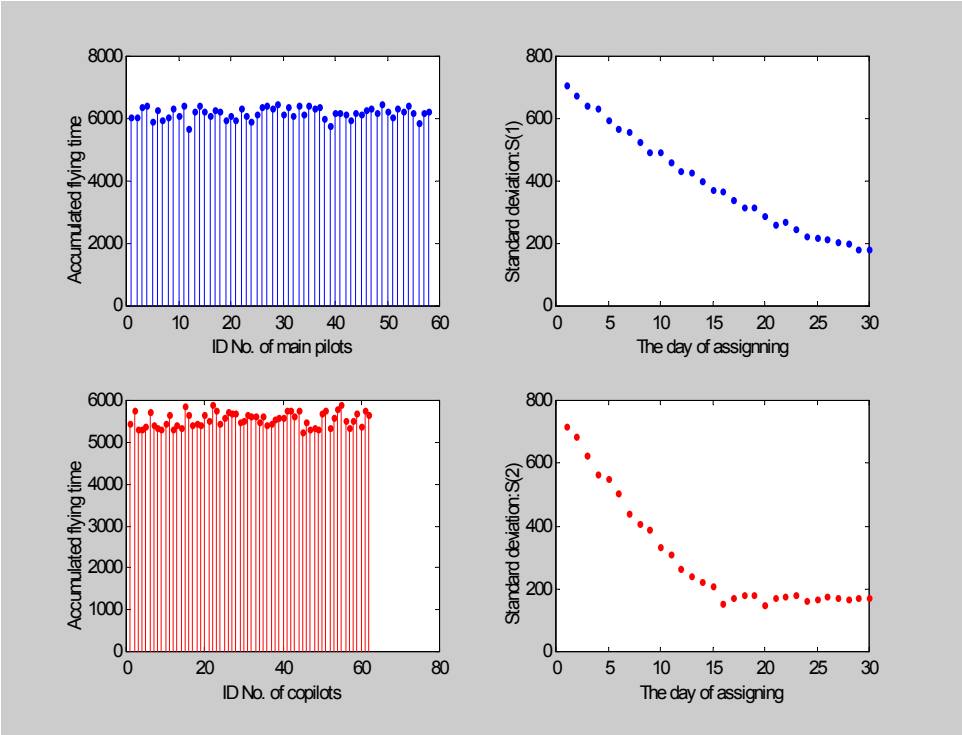

Figure 4.4.2 Computational result 1 (after 30th days' assignment)

Table 4.4.3 shows the change of the result when we make a few changes to the initial condition of tasks performing state while maintaining the initial distribution of accumulated flying time of the 58 main pilots and 62 copilots.

Figure 4.4.3 shows the initial condition. Also there is no much pilots redundancy. Figure 4.4.4 shows that both the s_1 and the s_2 are controlled very well, given the value (284.1) of control bound (4.30). However, the control effect of s_1 is not as good as the previous case. This indicates the influence of the initial state of tasks performing.

Table 4.4.3: Computational result 2 (m_1=58, m_2=62)

The number of pilots		Main pilots: 58	Copilots: 62
At the beginning	The standard deviation of accumulated flying time (minutes)	741.16	739.82
The maximum standard deviation of Accumulated flying time (minutes)		741.16	739.82
In the end (30^{th} day)	The standard deviation of accumulated flying time (minutes)	215.70	167.67
CPU (Intel Celeron 900) time (in minutes)		3.09	
The control bound: $\frac{\max_k T_k}{2\sqrt{2}}$		284.61	

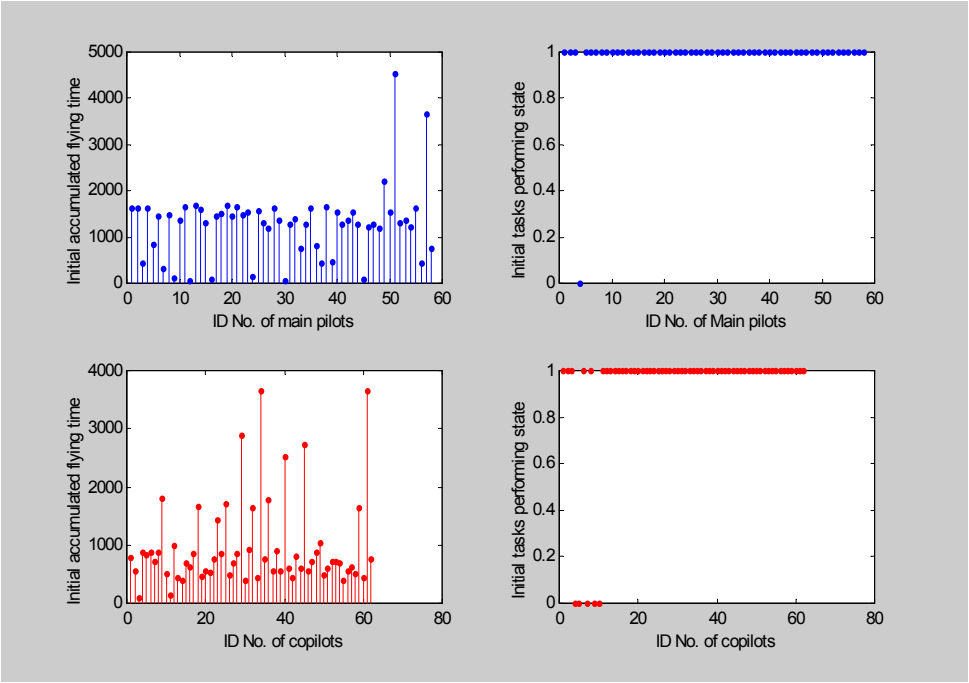

Figure 4.4.3 Initial condition 2 (m_1=58, m_2=62)

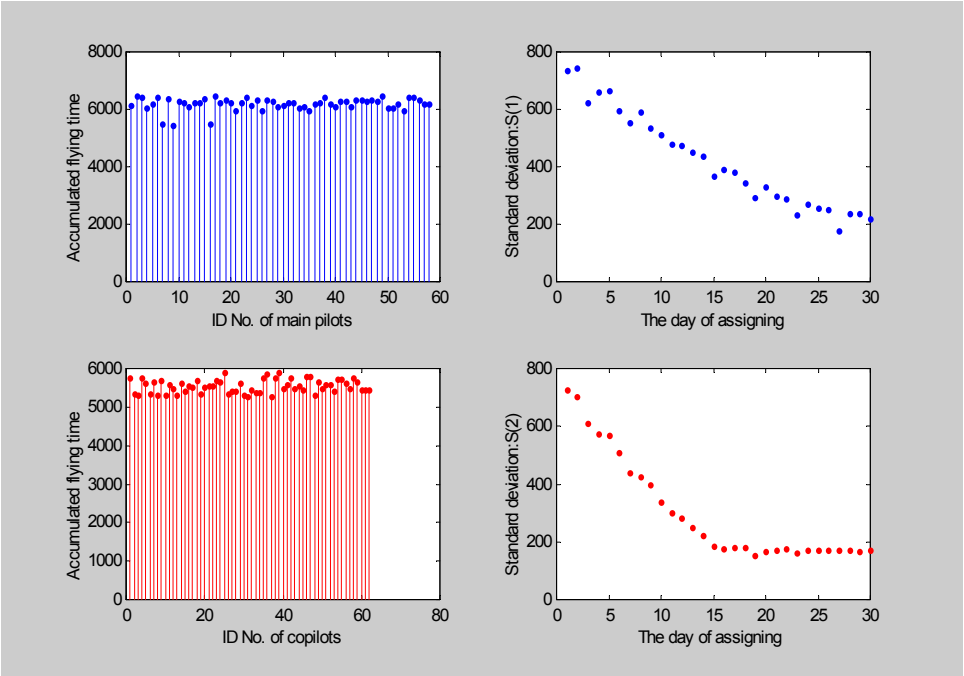

Figure 4.4.4 Computational result 2 (after 30th days' assignment)

Here with the same monthly flight tasks table, let's reduce the number of the pilots to 57 main pilots and 58 copilots. We maintain the same nonmatched relation between the main pilots and the copilots as Table 4.4.1. We will check what effect reducing the number of pilots will have.

Figure 4.4.5 shows that there is no main pilots redundancy because every main pilot is performing the task before the new tasks will be assigned. Figure 4.4.6 shows a fluctuation in the two standard deviations, and the maximum values appear not at the beginning. The standard deviations s_1 are not controlled well given the value (284.1) of control bound (4.30). This indicates the negative influence of reducing the number of pilots into minimum.

Table 4.4.4: Computational result 3 (m_1=57, m_2=58)

The number of pilots				Main pilots: 57	Copilots: 58
At the beginning	The standard deviation of accumulated flying time (minutes)			747.44	764.09
The maximum standard deviation of Accumulated flying time (minutes)				766.57	771.43
In the end	The standard deviation of accumulated flying time (minutes)	Assigning Day	31^{th}	363.38	179.65
			40^{th}	274.67	165.51
			56^{th}	223.76	187.25
			68^{th}	165.28	167.00
CPU (Intel Celeron 900) time (in minutes)		Assigning Day	31^{th}	3.07	
			40^{th}	3.92	
			56^{th}	4.57	
			68^{th}	6.94	
The control bound: $\frac{\max_k T_k}{2\sqrt{2}}$				284.61	

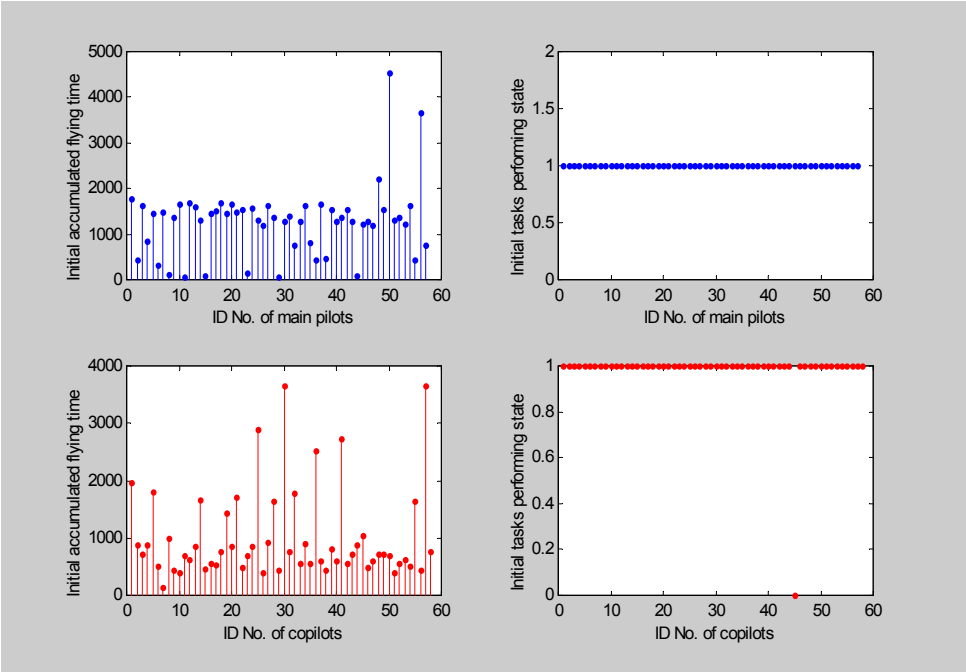

Figure 4.4.5 Initial condition 3 (m_1=57, m_2=58)

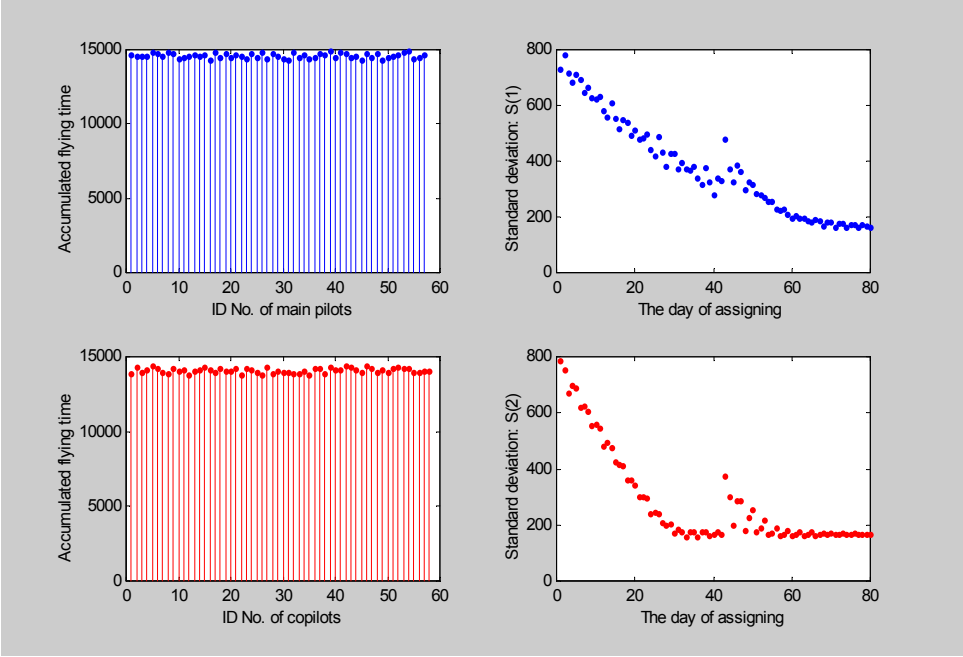

Figure 4.4.6 Computational result 3 (after 80th days' assignment)

Table 4.4.5 shows the change of the result when we make a few changes to the initial condition of tasks performing state while maintaining the initial distribution of accumulated flying time as Figure 4.4.5.

Figure 4.4.7 shows the initial condition. Also there is no main pilots redundancy. Figure 4.4.8 shows a fluctuation in the two standard deviations, and the maximum values appear not at the beginning. The standard deviations s_1 are not controlled well given the value (284.1) of control bound (4.30). This indicates the influence of reducing the number of pilots into minimum. The influence of the initial state of tasks performing plus the number of main pilots is significant. If we want better control, we may need more main pilots.

Table 4.4.5: Computational result 4 (m_1=57, m_2=58)

The number of pilots				Main pilots: 57	Copilots: 58
At the beginning	The standard deviation of accumulated flying time (minutes)			747.44	764.09
The maximum standard deviation of Accumulated flying time (minutes)				761.73	782.80
In the end	The standard deviation of accumulated flying time (minutes)	Assigning Day	34^{th}	329.39	180.64
			43^{th}	252.52	165.58
			49^{th}	233.89	168.80
			76^{th}	153.12	157.20
CPU (Intel Celeron 900) time (in minutes)		Assigning Day	34^{th}	3.49	
			43^{th}	4.02	
			49^{th}	4.35	
			76^{th}	7.18	
The control bound: $\frac{\max_k T_k}{2\sqrt{2}}$				284.61	

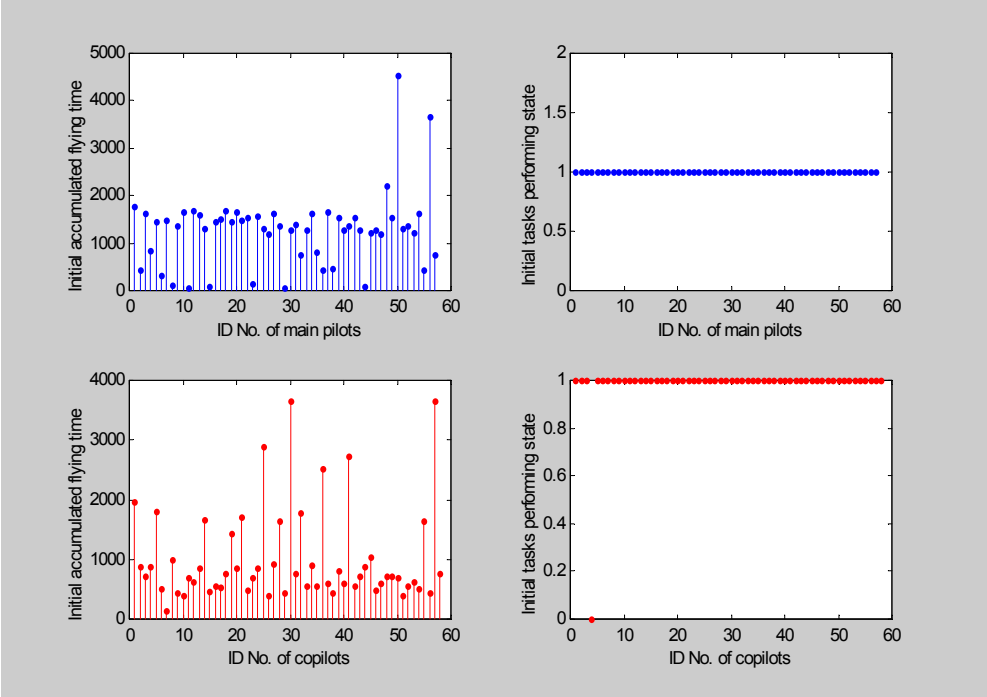

Figure 4.4.7 Initial condition 4 (m_1=57, m_2=58)

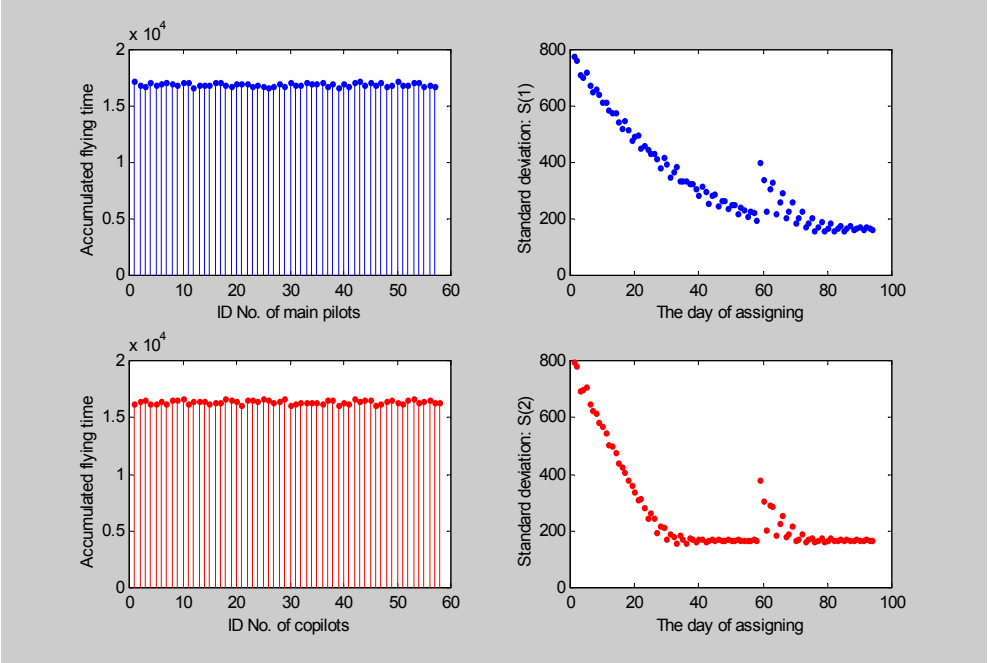

Figure 4.4.8 Computational result 4 (after 95th days' assignment)

From the numerical cases above, we can find that our algorithm compromises the negative influence of the nonmatched relation between the main pilots and the copilots. The reason is very simple; it is because of the sparse nonmatched relation between the main pilots and the copilots. So, under the assumption of sparse nonmatched relation between the main pilots and the copilots, our algorithm is very effective in controlling both s_1 and s_2.

Here we will check the negative effect of the nonmatched relation in the tasks assigning process if we add more nonmatched relation, In order to do this, we also need to eliminate the effects from other factors. So we will let the group of main pilots and the group of copilots have the same initial condition of accumulated flying time, the same

initial condition of tasks performing state, and the same number of the pilots. Based on the Table 4.4.1, we add more nonmatched relation and we let the number of both main pilots and copilots be 59.

Table 4.4.7 shows that s_1 and s_2 are both under well control given the value (284.1) of control bound (4.30).

Figure 4.4.10 also shows very similar descending ways s_1 and s_2 take, respectively. The negative influence of nonmatched relation here is not significant. This result strongly supports the effectiveness of our heuristic method. The nonmatched relation like what Table 4.4.6 shows is very uncommon in real world. However, its negative influence can be compromised well by our heuristic method.

Table 4.4.6: Nomathced relation 2 (m_1=59, m_2=59)

ID No. of the main pilots	ID No. of the nonmatched copilots
3	65, 73, 81
5	62, 63, 64, 65
9	71, 77, 80, 81
18	87,93
33	72, 75, 87
39	97,112
41	67,115
52	72, 75

Table 4.4.7: Computational result 5 (m_1=59, m_2=59)

<table>
<tr><td colspan="4">The number of pilots</td><td>Main pilots: 59</td><td>Copilots: 59</td></tr>
<tr><td colspan="2">At the beginning</td><td colspan="2">The standard deviation of accumulated flying time (minutes)</td><td>813.36</td><td>813.36</td></tr>
<tr><td colspan="4">The maximum standard deviation of Accumulated flying time (minutes)</td><td>813.35</td><td>813.36</td></tr>
<tr><td rowspan="2">In the end</td><td rowspan="2">The standard deviation of accumulated flying time (minutes)</td><td rowspan="2">Assigning Day</td><td>30th</td><td>173.32</td><td>179.91</td></tr>
<tr><td>40th</td><td>166.85</td><td>167.07</td></tr>
<tr><td colspan="2" rowspan="2">CPU (Intel Celeron 900) time (in minutes)</td><td rowspan="2">Assigning Day</td><td>34th</td><td colspan="2">3.45</td></tr>
<tr><td>40th</td><td colspan="2">4.11</td></tr>
<tr><td colspan="4">The control bound: $\frac{\max_k T_k}{2\sqrt{2}}$</td><td colspan="2">284.61</td></tr>
</table>

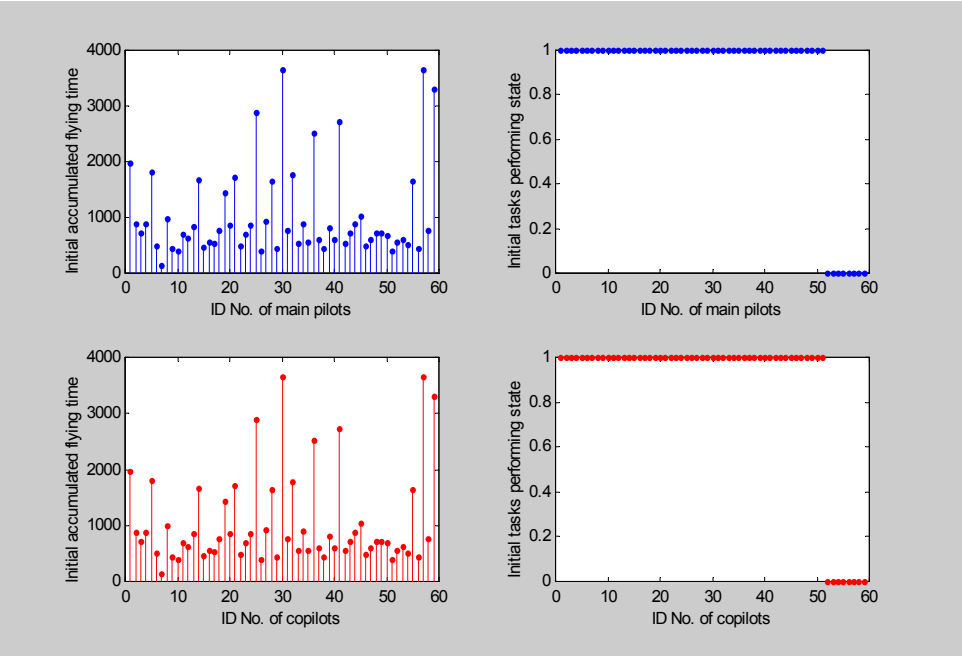

Figure 4.4.9 Initial condition 5 (m_1=59, m_2=59)

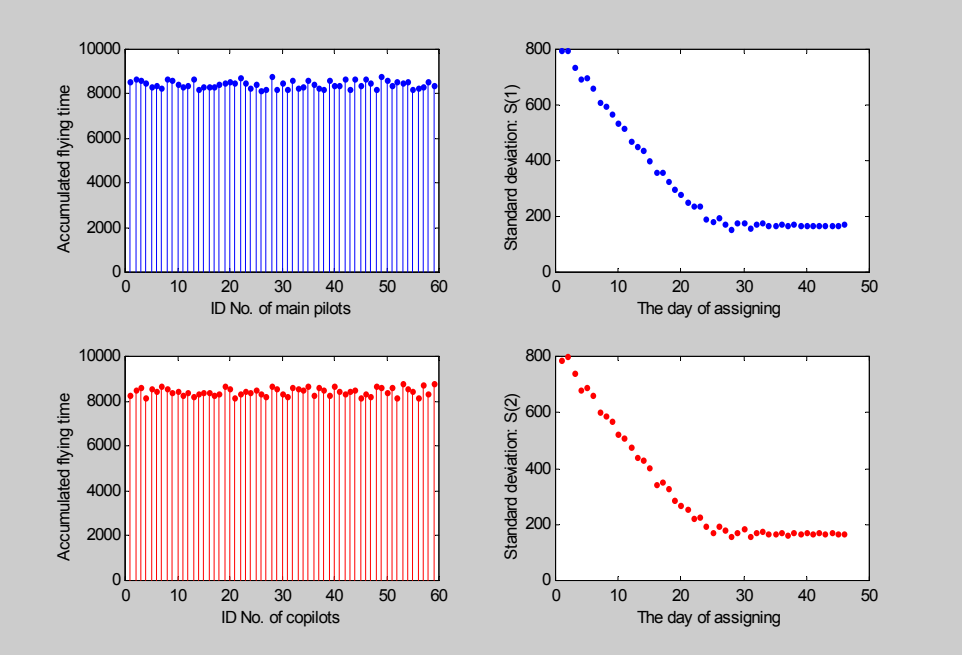

Figure 4.4.10 Computational result 5 after 46th days' assignment

CHAPTER 5

CONCLUSION

CONCLUSION AND PROSPECT

The numerical case of the model (1.1-1.12) in our computation is large scale because of the tens of thousands of 0-1 variables and millions of constraints. In the numerical test, our heuristic algorithm did good job because not only it controls well the deviation s_1 in (1.2) and s_2 in (1.7) (this means the satisfactory solution) while satisfying the constraints (1.3-1.6) and (1.8-1.11), as well as (1.12) but also it does this within polynomial time (minutes in our instances). The heuristic algorithm is effective and applicable in the real moderate large-scale rostering problems.

Based on the practical point of view, the assumption of sparse nonmatched relation between the main pilots and the copilots is reasonable. To some degree, our heuristic algorithm can compromise the influence of main-pilots-preferred principle well. This means that not only we can control s_1 under the upper bound (4.30) but also we can control s_2 very well, as the result of the design of weights setting before solving the phase-based weighted matching problem.

Also, we find that, when given a period of flight tasks table, we need enough pilots to fulfill the jobs. That is because not only we must ensure the accomplishment of the flight tasks, but also we want as good control over s_1 and s_2 as possible. The more pilots we

have, the more freedom we have to realize the control. However, to increase pilots' number means the increase in the cost the Airline Company may pay. So, the balance between needs to be found. And, promisingly, the numerical tests give a good support for deciding the appropriate construction of the pilot team.

The real flight crew assigning is far more complex than what the model (1.1-1.12) describes. We need to take into consideration far more factors: the privilege of some flight tasks, the positions exchange between some main pilots and some copilots, some irregular changes in the given flight tasks table, some personal events or preference involved etc. Also the complexity is far more than that: some international flight tasks need more than just one pair of pilots (we call this case the task overlap), we may have more than just one pilots base so that we must consider the cases of multi-bases crew assignment problem. Any additional consideration can add fearful scale to the existing complexity. And it is not strange that more and more models of large scale NP nonlinear integer programming arise. The biggest challenge that traditional techniques in optimization face is the unacceptable computational time, given the considerable scale of the problems. But the practical application emphasizes more on satisfactory solution plus acceptable computational time than optimal solution plus unacceptable time of computation. In this point, the modern heuristic algorithms in optimization, although still immature in theory, will surely win their wings in the modern industry.

APPENDIX

IMPLEMENT OF THE HEURISTIC METHOD

USING C++ AND MATLAB

C++ VERSION:

```
//#include "c:\user\yanghua\zhou\yxg\yxg\data.h"
//#include <stdio.h>
//#include <math.h>
#include <string.h>
//#include <iostream.h>
//#include <fstream.h>

//#include "c:\user\yanghua\zhou\yxg\yxg\data.h"
//#include "c:\user\yanghua\zhou\yxg\yxg\function.h"
#include <fstream.h>
#include <iostream.h>
#include <stdio.h>
#include <math.h>
#include <process.h>
#define MAXVEX 36
#define INFINITY 10000

struct TaskData{
        int year;
        int month;
        int day;
}  TaskTime[101];

int DayNumber;

int PilotOneNumber;/* the number of skippers */
int PilotTwoNumber;/* the number of copilots */
```

```
float AdjMatrix[MAXVEX][MAXVEX];
int  ShortPathWayMatrix[MAXVEX][MAXVEX];

int N;
int s;
int t;

int NoMatchNumber;

struct PilotData{
        char * Name;
        int rank;/* if skipper then rank=1;if copilot then rank=2 */
        float t;
        int FT;
};

struct PilotGroupCell{
        struct PilotData * PilotOne;
        struct PilotData * PilotTwo;
};

struct nomatch{         /* the nomatch pilot part */
        char * PilotOne; /* the name of skippers */
        char * PilotTwo; /* the name of copilots */
};

struct Task{
        char * Number;/* the number of flight */
        char * Name;/* the name of flight */
        int BeginTimeYear;
        int BeginTimeMon;
        int BeginTimeDay;
        int EndTimeYear;
        int EndTimeMon;
        int EndTimeDay;
        float ft;
        int FT;
        struct PilotGroupCell PilotGroup;/* the pilot parts of this task */
};
```

```
void GetDataFile(ofstream &cf_Pilot_w,ofstream &cf_Task_w,ofstream &cf_Match_w);

float shortpath(int * PathPrenode);

void GetShortPathWayMatrix(int * PathPrenode);

int GetTodayTasks(ifstream &cf_Task_r, struct Task * TK,struct TaskData Date);
void GetPilots(ifstream &cf_Pilot_r,struct PilotData * PILOT);
void BubbleSort(struct PilotData * array,int size,int tag);
void BubbleSortTK(struct Task * array,int size,int tag);
void GetPilotGroups(struct PilotGroupCell * PilotGroups,struct PilotData * PILOT);
void GetNoMatch(ifstream &cf_Match_r,struct nomatch * NoMatch);
void GetAdjMatrix(struct PilotData * PILOT,struct nomatch * NoMatch);
void GetTaskTable(int TaskNumber,struct Task * TK,ofstream &cf_TaskTable_w);
void PrintTaskAllocation(ifstream &cf_TaskTable_r);

void main()
{
        int  PathPrenode[MAXVEX];

        int counter;
        int i,j;

        int TaskNumber;/* the number of today'stasks */
        int day;/* the day counter */
        float shortest;/* the total cost or total weigh of the shortest path. */

        ofstream cf_Pilot_w("c:\\user\\yanghua\\zhou\\yxg\\yxg\\PilotMessage.dat");/* pilots'message */
        if(!(cf_Pilot_w))
          printf("the file could not be opened.\n");
        ifstream cf_Pilot_r("c:\\user\\yanghua\\zhou\\yxg\\yxg\\PilotMessage.dat");/* pilots'message */

        ofstream cf_Task_w("c:\\user\\yanghua\\zhou\\yxg\\yxg\\TaskMessage.dat");
        ifstream cf_Task_r("c:\\user\\yanghua\\zhou\\yxg\\yxg\\TaskMessage.dat");

        ofstream cf_Match_w("c:\\user\\yanghua\\zhou\\yxg\\yxg\\NoMatch.dat");
        ifstream cf_Match_r("c:\\user\\yanghua\\zhou\\yxg\\yxg\\NoMatch.dat");
```

```
ofstream cf_TaskTable_w("c:\\user\\yanghua\\zhou\\yxg\\yxg\\TaskTable.dat");
ifstream cf_TaskTable_r("c:\\user\\yanghua\\zhou\\yxg\\yxg\\TaskTable.dat");

struct Task * TaskTable;
struct PilotData * PilotSort;
struct Task * TK;/* today's tasks */
struct PilotData  PILOT[MAXVEX];/* the table of pilots*/
struct PilotGroupCell * PilotGroups;/* today's PilotGroups */
struct nomatch * NoMatch;/* nomatch message */

GetDataFile(cf_Pilot_w,cf_Task_w,cf_Match_w);
GetPilots(cf_Pilot_r,PILOT);
for(counter=PilotOneNumber+1;counter<=N-2;counter++)
   PilotSort[counter-PilotOneNumber]=PILOT[counter];
BubbleSort(PILOT,PilotOneNumber,2);
BubbleSort(PilotSort,PilotOneNumber,1);
for(counter=1;counter<=PilotTwoNumber;counter++)
   PILOT[PilotOneNumber+1]=PilotSort[counter];
/* We get PILOT matrix. */

GetNoMatch(cf_Match_r,NoMatch);
/* We get the number of nonmatch Pilot parts. */

for(day=1;day<=DayNumber;day++)
  {
    TaskNumber=GetTodayTasks(cf_Task_r,TK,TaskTime[day]);
    GetAdjMatrix(PILOT,NoMatch);
    counter=0;
    while(counter<TaskNumber)
           {
             counter=0;
             shortest=shortpath(PathPrenode);
             if(shortest>=INFINITY)
               {
                     printf("Have not enough parts of pilots\n");
                     exit;
               }/* end if */
             GetShortPathWayMatrix(PathPrenode);
             for(i=1;i<=PilotOneNumber;i++)
               {
                     if(ShortPathWayMatrix[s][i])
```

```
                                AdjMatrix[s][i]=INFINITY;
                                for(j=PilotTwoNumber+1;j<=N-2;j++)
                                  {
                                        if(ShortPathWayMatrix[i][j])
                                         {
                                           counter++;
                                           AdjMatrix[j][i]=(-1)*AdjMatrix[i][j];
                                           AdjMatrix[i][j]+=INFINITY;
                                         }/* end if */
                                        if(ShortPathWayMatrix[j][t])
                                        AdjMatrix[j][t]=INFINITY;
                                  }/* end for */
                               }/* end for */
                        }/* end while */
                GetPilotGroups(PilotGroups,PILOT);
                BubbleSortTK(TK,TaskNumber,2);
                for(i=1;i<=TaskNumber;i++)
                        {
                          TK[i].PilotGroup=PilotGroups[i];
                          PilotGroups[i].PilotOne->t+=TK[i].ft;
                          PilotGroups[i].PilotTwo->t+=TK[i].ft;
                          PilotGroups[i].PilotOne->FT=TK[i].FT;
                          PilotGroups[i].PilotTwo->FT=TK[i].FT;
                        }/* end for */
                GetTaskTable(TaskNumber,TK,cf_TaskTable_w);
                /* put today's tasks into TaskTable*/
                for(i=1;i<=N-2;i++)
                        {
                          if(PILOT[i].FT)PILOT[i].FT--;
                        }/* end for */
              }/* end for */
            PrintTaskAllocation(cf_TaskTable_r);
}/* end main() */

/*********************************************************************/

void BubbleSortTaskData(struct TaskData * array,
                        int size)
{
   struct TaskData temp;
   int pass,j;
```

```
   for(pass=1;pass<=size-1;pass++)
   for(j=size;j>=pass+1;j--)
   {
            if(array[j].year<array[j-1].year)
              {
                temp=array[j];
                array[j]=array[j-1];
                array[j-1]=temp;
              }/* end if */
            else
              {
               if(array[j].month>array[j-1].month)
                 {
                        temp=array[j];
                        array[j]=array[j-1];
                        array[j-1]=temp;
                 }/* end if */
               else
                  {
                        if(array[j].day>array[j-1].day)
                          {
                            temp=array[j];
                            array[j]=array[j-1];
                            array[j-1]=temp;
                          }/* end if */
                        }/* end else */
            }/* end else */
   }/* end for */
}/* end BubbleSortTaskData */

/*********************************************************************/

void GetDataFile(ofstream &cf_Pilot_w,
                        ofstream &cf_Task_w,
                        ofstream &cf_Match_w)
{
  int counter=1;/* "counter" and "i" used in input TaskTime[101] */
  int i;
```

```
char c=' ';   /* help put the data into files. */

struct PilotData pilot;
struct Task task;
struct nomatch NoMatch;
int tag; /* if tag=0 then input of the file is over. */

if(!cf_Pilot_w)
  printf("PilotMessage.dat could not be opened.\n");
else
 {
        printf("Enter the data of pilots.\n");
        printf("when 'the tag of skipper'=0,the input of tasks is over.\n");
        printf("Enter the tag of skipper.\n");
        cin>>tag;
        while(tag)
         {
           printf("Enter name of the pilot.\n");
           cin>>*(pilot.Name);
           printf("if skipper then rank=1,else rank=2.\n");
           printf("Enter the rank of pilots.\n");
           cin>>pilot.rank;
           printf("Enter total fiy time by hundred hours\n");
           cin>>pilot.t;
           cf_Pilot_w<<pilot.Name
                     <<c
                     <<pilot.rank
                     <<c
                     <<pilot.t
                     <<endl;
           cin>>tag;
         }/* end while */
 }/* end else */

if(!cf_Task_w)
  printf("TaskMessage.dat could not be opened.\n");
else
 {
   printf("when 'the tag of task'=0,the input of tasks is over.\n");
   printf("Enter the tag of fligher.\n");
   cin>>tag;
```

```
while(tag)
        {
         printf("Enter airline name,beignning time,end time of task \n?");
         cin>>*(task.Number);
         printf("Enter the name of the airline of the task.\n");
         cin>>*(task.Name);
         printf("Enter the year of the beignning time of the task.\n");
         cin>>task.BeginTimeYear;
         printf("Enter the month of the beignning time of the task.\n");
         cin>>task.BeginTimeMon;
         printf("Enter the day of the beignning time of the task.\n");
         cin>>task.BeginTimeDay;
         printf("Enter the year of the end time of the task.\n");
         cin>>task.EndTimeYear;
         printf("Enter the month of the end time of the task.\n");
         cin>>task.EndTimeMon;
         printf("Enter the day of the end time of the task.\n");
         cin>>task.EndTimeDay;
         printf("Enter the fly time of the task by hundred hours.\n");
         cin>>task.ft;
         printf("Enter the number of days of the task.\n");
         cin>>task.FT;
         cf_Task_w<<*(task.Number) /* put the record of the task into file */
                  <<c
                  <<*(task.Name)
                  <<c
                  <<task.BeginTimeYear
                  <<c
                  <<task.BeginTimeMon
                  <<c
                  <<task.BeginTimeDay
                  <<c
                  <<task.EndTimeYear
                  <<c
                  <<task.EndTimeMon
                  <<c
                  <<task.EndTimeDay
                  <<c
                  <<task.ft
                  <<c
                  <<task.FT
```

```
                    <<endl;
            for(i=1;i<=counter;i++)
              {
                    if(TaskTime[i].year==task.BeginTimeYear&&
                      TaskTime[i].month==task.BeginTimeMon&&
                      TaskTime[i].day==task.BeginTimeDay)
                      break;
               }/* end for */
            if(i==counter+1)/* Put the new beignning time of task into TsakTime[101] */
              {
                    counter=i;
                    TaskTime[counter].year=task.BeginTimeYear;
                    TaskTime[counter].month=task.BeginTimeMon;
                    TaskTime[counter].day=task.BeginTimeDay;
              }/* end if */
            printf("when 'the tag of task'=0,the input of tasks is over.\n");
            printf("Enter the tag of fligher.\n");
            cin>>tag;
          }/* end while */
  DayNumber=counter;
  BubbleSortTaskData(TaskTime,DayNumber);          /* use bubblesort for TaskTime[] */
 }/* end else */


if(!(cf_Match_w))
  printf("NoMatch.dat could not be opened.\n");
else
  {
          printf("Enter the tag of the nomatch.\n");
          printf("when 'the tag of nomatch'=0,the input of task is over.\n");
          cin>>tag;
          while(tag)
            {
              printf("Enter the name of the skipper of the nomatch parts.\n?");
              cin>>*(NoMatch.PilotOne);
              printf("Enter the name of the copilot of the nomatch part.\n");
              cin>>*(NoMatch.PilotTwo);
              cf_Match_w<<NoMatch.PilotOne
                        <<c
                        <<NoMatch.PilotTwo
                        <<endl;
```

```
                cin>>tag;
              }/* end while */
    }/* end else */
}/* end  GetDataFile */


/************************************************************/

float shortpath(int * PathPrenode)
{
   int Final[MAXVEX];
   int       w;
   int FinalNode;
   int NumOfFinal;
   float MinPath;
   float dist[MAXVEX];
   for(w=1;w<=N;w++)
           PathPrenode[w]=0;
   for(w=1;w<=N;w++)
     {
            dist[w]=AdjMatrix[s][w];
            if(AdjMatrix[s][w]<INFINITY)
            PathPrenode[w]=s;
     }/* end for */
   for(w=1;w<=N;w++)
   Final[w]=0;
   Final[s]=1;
   NumOfFinal=1;
   while(NumOfFinal<N-1)
     {
            MinPath=dist[w];
            FinalNode=s;
            for(w=1;w<=N;w++)
              {
                    if(!Final[w]&&dist[w]<MinPath)
                      {
                        FinalNode=w;
                        MinPath=dist[w];
                      }/* end if */
              }/* end for */
            Final[FinalNode]=1;
```

```
            NumOfFinal++;
            for(w=1;w<=N;w++)
              {
               if(!Final[w]&&dist[FinalNode]+
                        AdjMatrix[FinalNode][w]<dist[w])
                        {
                          dist[w]=dist[FinalNode]+AdjMatrix[FinalNode][w];
                          PathPrenode[w]=FinalNode;
                        }/* end if */
              }/* end for */
      }/* end while */
      return dist[N];
}/* end shortpath */

/************************************************************/
void GetShortPathWayMatrix(int * PathPrenode)

{
  int k;
  int w;
  PathPrenode[s]=0;
  k=t;
  while(PathPrenode[k])
      {
            w=PathPrenode[k];
            ShortPathWayMatrix[w][k]=1;/*ShortPathWayMatrix[w][k]*/
      }/* end while */
}/* end GetShortPathWayMatrix */

/**********************************************************/

int GetTodayTasks(ifstream &cf_Task_r,
                        struct Task * TK,     /* today's tasks */
                        struct TaskData Date) /* the date of today */
{
  int k;/* today's number of tasks */

  char c;/* help output the date from cf_Task. */

  if(!(cf_Task_r))
  printf("TaskMessage.dat could not be opened.\n");
```

```
  else
    {
            k=0;
            while(cf_Task_r)
              {
                cf_Task_r>>TK[k+1].Number>>c
                                >>TK[k+1].Name>>c
                                >>TK[k+1].BeginTimeYear>>c
                                >>TK[k+1].BeginTimeMon>>c
                                >>TK[k+1].BeginTimeDay>>c
                                >>TK[k+1].EndTimeYear>>c
                                >>TK[k+1].EndTimeMon>>c
                                >>TK[k+1].EndTimeDay>>c
                                >>TK[k+1].ft>>c
                                >>TK[k+1].FT;
                if(TK[k+1].BeginTimeYear==Date.year&&
                        TK[k+1].BeginTimeMon==Date.month&&
                        TK[k+1].BeginTimeDay==Date.day)
                        k++;
              }/* end while */
    }/* end else */
   return k;/* today's number of tasks */
}/* end GetNoMatch*/

/************************************************************/

void GetPilots(ifstream &cf_Pilot_r,
                struct PilotData * PILOT)
{
   char c;  /* help output the data from cf_Pilot. */

   PilotOneNumber=0;
   N=0;
   if(!(cf_Pilot_r))
     printf("PilotMessage.dat could not be opened.\n");
   else
     {
            while(!(cf_Pilot_r))
              {
                        N++;
                        cf_Pilot_r>>PILOT[N].Name>>c
```

```
                                >>PILOT[N].rank>>c
                                >>PILOT[N].rank>>c
                                >>PILOT[N].t;
                        if(PILOT[N].rank==1)
                        PilotOneNumber++;
                    }/* end while */
                PilotTwoNumber=N-PilotOneNumber;
                N=N+2;
                s=N-1;
                t=N;
        }/* end else */
}/* end GetPilots */

/************************************************************/

void BubbleSort(struct PilotData * array,
                                        int size,
                                        int tag)
/* when tag=1,from little to big,
   when tag=2,from big to little. */
{
    struct PilotData temp;
    int tag1;
    int pass,j;
    tag1=tag;
    if(tag1==1)
            {
              for(pass=1;pass<=size-1;pass++)
              for(j=size;j>=pass+1;j--)
              if(array[j].t<array[j-1].t)
                {
                        temp=array[j];
                        array[j]=array[j-1];
                        array[j-1]=temp;
                }/* end if */
            }/* end if */
    else
            {
              for(pass=1;pass<=size-1;pass++)
              for(j=size;j>=pass+1;j--)
              if(array[j].t>array[j-1].t)
```

```
                        {
                         temp=array[j];
                         array[j]=array[j-1];
                         array[j-1]=temp;
                        }/* end if */
            }/* end else */
}/* end BubbleSort */

/*********************************************************/

void BubbleSortTK(struct Task * array,
                        int size,
                        int tag)
/* when tag=1,from little to big,
   when tag=2,from big to little. */
{
   struct Task temp;
   int tag1;
   int pass,j;
   tag1=tag;
   if(tag1==1)
            {
              for(pass=1;pass<=size-1;pass++)
              for(j=size;j>=pass+1;j--)
              if(array[j].ft<array[j-1].ft)
                {
                        temp=array[j];
                        array[j]=array[j-1];
                        array[j-1]=temp;
                }/* end if */
            }/* end if */
   else
            {
              for(pass=1;pass<=size-1;pass++)
              for(j=size;j>=pass+1;j--)
              if(array[j].ft>array[j-1].ft)
                {
                        temp=array[j];
                        array[j]=array[j-1];
                        array[j-1]=temp;
                }/* end if */
```

```
            }/* end else */
}/* end BubbleSort */

/************************************************************/

void GetPilotGroups(struct PilotGroupCell * PilotGroups,
                            struct PilotData * PILOT)
{
   int i,j;
   int k=0;
   for(i=1;i<=PilotOneNumber;i++)
            {
              for(j=PilotOneNumber+1;j<=PilotOneNumber+
                          PilotTwoNumber;j++)
                 {
                        if(ShortPathWayMatrix[i][j])
                          {
                            k++;
                            PilotGroups[k].PilotOne=&(PILOT[i]);
                            PilotGroups[k].PilotTwo=&(PILOT[j]);
                          }/* end if */
                 }/* end for */
            }/* end for */
}/* end GetPilotGroups */

/************************************************************/

void GetNoMatch(ifstream &cf_Match_r,
                        struct nomatch * NoMatch)
{

   char c;/* help output the data from cf_Match. */
   if(!(cf_Match_r))
     printf("NoMatch.dat could not be opened.\n");
   else
     {
              NoMatchNumber=0;
              while(!(cf_Match_r))
                {
                        NoMatchNumber++;
                        cf_Match_r>>NoMatch[NoMatchNumber].PilotOne
```

```
                                >>c
                                >>NoMatch[NoMatchNumber].PilotTwo;
            }/* end while */
    }/* end else */
}/* end GetNoMatch*/

/*********************************************************/

void GetAdjMatrix(struct PilotData * PILOT,
                  struct nomatch * NoMatch)
{
   int i,j,ii,jj;
   for(i=1;i<=N;i++)
     {
          for(j=1;j<=N;j++)
            {
             AdjMatrix[i][j]=INFINITY;
            }/* end for */
     }/* end for */
   for(i=1;i<=PilotOneNumber;i++)
     {
          AdjMatrix[s][i]=0;
          for(j=PilotOneNumber+1;j<=N-2;j++)
            {
             AdjMatrix[i][j]=fabs(PILOT[i].t-PILOT[j].t);
             AdjMatrix[j][t]=0;
            }/* end for */
     }/* end for */
   if(PilotOneNumber<=PilotOneNumber)
     {
         for(i=1;i<=PilotOneNumber;i++)
           {
            AdjMatrix[i][PilotOneNumber+i]=0;
           }/* end for */
     }/* end if */
   else
     {
          for(i=1;i<=PilotTwoNumber;i++)
            {
                AdjMatrix[i][PilotOneNumber+i]=0;
            }/* end for */
```

```
            i=INFINITY;
            jj=0;
            for(j=(3*PilotOneNumber+1)/2;
               j<=N-(2*PilotOneNumber-1)/2;j++)
               {
                 if(i>fabs(PILOT[(PilotOneNumber+1)/2].t-PILOT[j].t))
                        {
                          i=fabs(PILOT[(PilotOneNumber+1)/2].t-PILOT[j].t);
                          jj=j;
                        }/* end if */
               }/* end for */
            AdjMatrix[(PilotOneNumber+1)/2][jj]=0;
     }/* end else */
   if(NoMatch[1].PilotOne&&NoMatch[1].PilotTwo)
     {
            ii=1;
            jj=PilotOneNumber+1;
            for(i=1;i<=NoMatchNumber;i++)
              {
                        while(strcmp(PILOT[ii].Name,NoMatch[i].PilotOne))ii++;
                        while(strcmp(PILOT[jj].Name,NoMatch[i].PilotTwo))jj++;
                        AdjMatrix[ii][jj]=AdjMatrix[ii][jj]+INFINITY;
              }/* end for */
     }/* end if */
}/* end Get AdjMatrix */

/*********************************************************/

void GetTaskTable(int TaskNumber,
                        struct Task * TK,
                        ofstream &cf_TaskTable_w)
{
  char c;  /* help put the data into cf_TaskTable. */
  int i;

  if(!(cf_TaskTable_w))
  printf("TaskTable.dat could not be opened.\n");
  else
    {
             for(i=1;i<=TaskNumber;i++)
             cf_TaskTable_w<<*(TK[i].Number)
```

```
                                <<c
                                <<*(TK[i].Name)
                                <<c
                                <<TK[i].BeginTimeYear
                                <<c
                                <<TK[i].BeginTimeMon
                                <<c
                                <<TK[i].BeginTimeDay
                                <<c
                                <<TK[i].EndTimeYear
                                <<c
                                <<TK[i].EndTimeMon
                                <<c
                                <<TK[i].EndTimeDay
                                <<c
                                <<TK[i].ft
                                <<c
                                <<TK[i].FT
                                <<c
                                <<TK[i].PilotGroup.PilotOne->Name
                                <<c
                                <<TK[i].PilotGroup.PilotTwo->Name
                                <<endl;
    }/* end else */
}/* end GetTaskTable */

/*****************************************************************/

void PrintTaskAllocation(ifstream &cf_TaskTable_r)
{
    char c; /* help output the data from cf_TaskTable. */
    struct Task task;

    if(!(cf_TaskTable_r))
    printf("PilotMessage.dat could not be opened.\n");
    else
            {
              while(!(cf_TaskTable_r))
                {
                        cf_TaskTable_r>>*(task.Number)
                                        >>c
```

```
                                        >>*(task.Name)
                                        >>c
                                        >>task.BeginTimeYear
                                        >>c
                                        >>task.BeginTimeMon
                                        >>c
                                        >>task.BeginTimeDay
                                        >>c
                                        >>task.EndTimeYear
                                        >>c
                                        >>task.EndTimeMon
                                        >>c
                                        >>task.EndTimeDay
                                        >>c
                                        >>task.ft
                                        >>c
                                        >>task.FT
                                        >>c
                                        >>task.PilotGroup.PilotOne->Name
                                        >>c
                                        >>task.PilotGroup.PilotTwo->Name;
                        cout<<task.Number
                           <<c
                           <<task.BeginTimeYear
                           <<c
                           <<task.BeginTimeMon
                           <<c
                           <<task.BeginTimeDay
                           <<c
                           <<task.EndTimeYear
                           <<c
                           <<task.EndTimeMon
                           <<c
                           <<task.EndTimeDay
                           <<c
                           <<task.PilotGroup.PilotOne->Name
                           <<c
                           <<task.PilotGroup.PilotTwo->Name;
                    }/* end while */
                }/* end else */
        }/* end PrintTaskAllocation */
```

Matlab VERSION:

Block1:

```
%generate the flight tasks table:

taskyear=2001;
taskmonth=5;
%%%%%%%%%%first cycle of the tasks

day_track=1;%pointer of days
%**********1******************
i=1;j=1;
task(i).number=i;%task tracking Number which is as the order of asending start time
a=[taskyear taskmonth day_track 7 0 0];
b=[taskyear taskmonth day_track+2 9 17 0];
task(i).starttime=a;
task(i).endtime=b;

[diff_t,diff_minutes]=diff_time(a,b);

task(i).tasktime=diff_minutes;
task(i).flyingtime=604;
task(i).pilot='not assigned';
task(i).copilot='not assigned';
%*****************************

%**********2******************
i=i+1;j=j+1;
task(i).number=i;%task tracking Number which is as the order of asending start time
a=[taskyear taskmonth day_track 7 30 0];
b=[taskyear taskmonth day_track+2 8 14 0];
task(i).starttime=a;
task(i).endtime=b;

[diff_t,diff_minutes]=diff_time(a,b);
```

```
task(i).tasktime=diff_minutes;
task(i).flyingtime=521;
task(i).pilot='not assigned';
task(i).copilot='not assigned';
%*******************************

%**********3******************
i=i+1;j=j+1;
task(i).number=i;%task tracking Number which is as the order of asending start time
a=[taskyear taskmonth day_track 8 15 0];
b=[taskyear taskmonth day_track+2 17 45 0];
task(i).starttime=a;
task(i).endtime=b;

[diff_t,diff_minutes]=diff_time(a,b);

task(i).tasktime=diff_minutes;
task(i).flyingtime=645;
task(i).pilot='not assigned';
task(i).copilot='not assigned';
%*******************************

%**********4******************
i=i+1;j=j+1;
task(i).number=i;%task tracking Number which is as the order of asending start time
a=[taskyear taskmonth day_track 9 50 0];
b=[taskyear taskmonth day_track+2 22 10 0];
task(i).starttime=a;
task(i).endtime=b;

[diff_t,diff_minutes]=diff_time(a,b);

task(i).tasktime=diff_minutes;
task(i).flyingtime=441;
task(i).pilot='not assigned';
task(i).copilot='not assigned';
%*******************************

%**********5******************
i=i+1;j=j+1;
```

```
task(i).number=i;%task tracking Number which is as the order of asending start time
a=[taskyear taskmonth day_track 10 30 0];
b=[taskyear taskmonth day_track+2 13 15 0];
task(i).starttime=a;
task(i).endtime=b;

[diff_t,diff_minutes]=diff_time(a,b);

task(i).tasktime=diff_minutes;
task(i).flyingtime=573;
task(i).pilot='not assigned';
task(i).copilot='not assigned';
%******************************

%**********6******************
i=i+1;j=j+1;
task(i).number=i;%task tracking Number which is as the order of asending start time
a=[taskyear taskmonth day_track 12 30 0];
b=[taskyear taskmonth day_track+2 23 0 0];
task(i).starttime=a;
task(i).endtime=b;

[diff_t,diff_minutes]=diff_time(a,b);

task(i).tasktime=diff_minutes;
task(i).flyingtime=396;
task(i).pilot='not assigned';
task(i).copilot='not assigned';
%******************************

%**********7******************
i=i+1;j=j+1;
task(i).number=i;%task tracking Number which is as the order of asending start time
a=[taskyear taskmonth day_track 14 15 0];
b=[taskyear taskmonth day_track+2 14 0 0];
task(i).starttime=a;
task(i).endtime=b;

[diff_t,diff_minutes]=diff_time(a,b);

task(i).tasktime=diff_minutes;
```

```
task(i).flyingtime=394;
task(i).pilot='not assigned';
task(i).copilot='not assigned';
%*******************************

%*********8******************
i=i+1;j=j+1;
task(i).number=i;%task tracking Number which is as the order of asending start time
a=[taskyear taskmonth day_track 14 55 0];
b=[taskyear taskmonth day_track+2 19 30 0];
task(i).starttime=a;
task(i).endtime=b;

[diff_t,diff_minutes]=diff_time(a,b);

task(i).tasktime=diff_minutes;
task(i).flyingtime=523;
task(i).pilot='not assigned';
task(i).copilot='not assigned';
%*******************************

%*********9******************
i=i+1;j=j+1;
task(i).number=i;%task tracking Number which is as the order of asending start time
a=[taskyear taskmonth day_track 15 45 0];
b=[taskyear taskmonth day_track+2 17 10 0];
task(i).starttime=a;
task(i).endtime=b;

[diff_t,diff_minutes]=diff_time(a,b);

task(i).tasktime=diff_minutes;
task(i).flyingtime=440;
task(i).pilot='not assigned';
task(i).copilot='not assigned';
%*******************************

%*********10******************
i=i+1;j=j+1;
task(i).number=i;%task tracking Number which is as the order of asending start time
a=[taskyear taskmonth day_track 16 20 0];
```

```
b=[taskyear taskmonth day_track+2 20 35 0];
task(i).starttime=a;
task(i).endtime=b;

[diff_t,diff_minutes]=diff_time(a,b);

task(i).tasktime=diff_minutes;
task(i).flyingtime=601;
task(i).pilot='not assigned';
task(i).copilot='not assigned';
%******************************


%**********11******************
i=i+1;j=j+1;
task(i).number=i;%task tracking Number which is as the order of asending start time
a=[taskyear taskmonth day_track 16 55 0];
b=[taskyear taskmonth day_track+2 11 30 0];
task(i).starttime=a;
task(i).endtime=b;

[diff_t,diff_minutes]=diff_time(a,b);

task(i).tasktime=diff_minutes;
task(i).flyingtime=382;
task(i).pilot='not assigned';
task(i).copilot='not assigned';
%******************************

%**********12******************
i=i+1;j=j+1;
task(i).number=i;%task tracking Number which is as the order of asending start time
a=[taskyear taskmonth day_track 17 10 0];
b=[taskyear taskmonth day_track+3 9 20 0];
task(i).starttime=a;
task(i).endtime=b;

[diff_t,diff_minutes]=diff_time(a,b);

task(i).tasktime=diff_minutes;
task(i).flyingtime=697;
```

```
task(i).pilot='not assigned';
task(i).copilot='not assigned';
%*******************************

%**********13******************
i=i+1;j=j+1;
task(i).number=i;%task tracking Number which is as the order of asending start time
a=[taskyear taskmonth day_track 18 40 0];
b=[taskyear taskmonth day_track+3 11 25 0];
task(i).starttime=a;
task(i).endtime=b;

[diff_t,diff_minutes]=diff_time(a,b);

task(i).tasktime=diff_minutes;
task(i).flyingtime=760;
task(i).pilot='not assigned';
task(i).copilot='not assigned';
%*******************************


%**********14******************
i=i+1;j=j+1;
task(i).number=i;%task tracking Number which is as the order of asending start time
a=[taskyear taskmonth day_track 19 15 0];
b=[taskyear taskmonth day_track+3 9 40 0];
task(i).starttime=a;
task(i).endtime=b;

[diff_t,diff_minutes]=diff_time(a,b);

task(i).tasktime=diff_minutes;
task(i).flyingtime=602;
task(i).pilot='not assigned';
task(i).copilot='not assigned';
%*******************************

%**********15******************
i=i+1;j=j+1;
task(i).number=i;%task tracking Number which is as the order of asending start time
a=[taskyear taskmonth day_track 20 40 0];
```

```
b=[taskyear taskmonth day_track+3 10 10 0];
task(i).starttime=a;
task(i).endtime=b;

[diff_t,diff_minutes]=diff_time(a,b);

task(i).tasktime=diff_minutes;
task(i).flyingtime=595;
task(i).pilot='not assigned';
task(i).copilot='not assigned';
%******************************

%**********16******************
i=i+1;j=j+1;
task(i).number=i;%task tracking Number which is as the order of asending start time
a=[taskyear taskmonth day_track 21 20 0];
b=[taskyear taskmonth day_track+3 14 50 0];
task(i).starttime=a;
task(i).endtime=b;

[diff_t,diff_minutes]=diff_time(a,b);

task(i).tasktime=diff_minutes;
task(i).flyingtime=506;
task(i).pilot='not assigned';
task(i).copilot='not assigned';
%******************************

%**********17******************
i=i+1;j=j+1;
task(i).number=i;%task tracking Number which is as the order of asending start time
a=[taskyear taskmonth day_track 22 30 0];
b=[taskyear taskmonth day_track+3 19 10 0];
task(i).starttime=a;
task(i).endtime=b;

[diff_t,diff_minutes]=diff_time(a,b);

task(i).tasktime=diff_minutes;
task(i).flyingtime=805;
task(i).pilot='not assigned';
```

```
task(i).copilot='not assigned';
%*******************************

%****display the information of tasks of the first task cycle
%for i1=1:i
%   task(i1),
%end

%begin of the whole task cycle
while day_track < a3_max(task(i).starttime)

  day_track=day_track + 1;%pointer of days

  for ii = i+1 : i+j

    task(ii).number=ii;%task tracking Number which is as the order of asending start time
    a=task(ii-j).starttime;
    a(3)=day_track;
    task(ii).starttime=a;

    day_increase=0;
    switch (ii-(i+1)+1)%**********************switch
    case 1
      day_increase=2;
    case 2
      day_increase=2;
    case 3
      day_increase=2;
    case 4
      day_increase=2;
    case 5
      day_increase=2;
    case 6
      day_increase=2;
    case 7
      day_increase=2;
    case 8
      day_increase=2;
    case 9
      day_increase=2;
    case 10
```

```
   day_increase=2;
case 11
   day_increase=2;
case 12
   day_increase=3;
case 13
   day_increase=3;
case 14
   day_increase=3;
case 15
   day_increase=3;
case 16
   day_increase=3;
case 17
   day_increase=3;

end%**********************end switch

b=task(ii-j).endtime;
b(3)=day_track + day_increase;


if b(3) > a3_max(a)
   b(3)=b(3)-a3_max(a);
   b(2)=a(2)+1;
end

if b(2)>12
   b(2)=b(2)-12;
   b(1)=a(1)+1;
end

task(ii).endtime=b;

[diff_t,diff_minutes]=diff_time(a,b);

task(ii).tasktime=diff_minutes;

c=task(ii-j).flyingtime;
task(ii).flyingtime=c;
```

```
    task(ii).pilot='not assigned';
    task(ii).copilot='not assigned';

  end%end for
  i=ii;

end%end while
%end of the whole task cycle
ii,

%for n=1:ii
 %  aa=task(n).starttime;
 % if  aa(3)== 3
 %     task(n),
 %  end
%end
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
function [diff_t,diff_minutes]=diff_time(a,b)
c=[0 0 0 0 0 0];%year,month,day,hour,minute,


%seconds
c(6)=0;

%minutes
if b(5)>= a(5)
  c(5)=b(5)-a(5);
else
  c(5)=b(5)-a(5)+60;
  b(4)=b(4)-1;
end
%hours
if b(4)>= a(4)
  c(4)=b(4)-a(4);
else
  c(4)=b(4)-a(4)+24;
  b(3)=b(3)-1;
end
%days
if b(2)== a(2)
  c(3)=b(3)-a(3);
```

```
else

  %janurary to feburary
  if b(2)==2 & a(2)==1
    c(3)=b(3)-a(3)+31;
  end

  %feburary to march
  if b(2)==3 & a(2)==2
    TU=0;
    if b(1)==a(1) & mod(a(1),4)<0.001
      TU=29;
    else
      TU=28;
    end

    c(3)=b(3)-a(3)+TU;
  end

  %March to April
  if b(2)==4 & a(2)==3
    c(3)=b(3)-a(3)+31;
  end

  %April to May
  if b(2)==5 & a(2)==4
    c(3)=b(3)-a(3)+30;
  end

  %May to June
  if b(2)==6 & a(2)==5
    c(3)=b(3)-a(3)+31;
  end

  %June to July
  if b(2)==7 & a(2)==6
    c(3)=b(3)-a(3)+30;
  end

  %July to August
  if b(2)==8 & a(2)==7
```

```
    c(3)=b(3)-a(3)+31;
  end

  %August to september
  if b(2)==9 & a(2)==8
    c(3)=b(3)-a(3)+31;
  end

  %September to October
  if b(2)==10 & a(2)==9
    c(3)=b(3)-a(3)+30;
  end

  %October to November
  if b(2)==11 & a(2)==10
    c(3)=b(3)-a(3)+31;
  end

  %November to December
  if b(2)==12 & a(2)==11
    c(3)=b(3)-a(3)+30;
  end

  %December to Janurary
  if b(2)==1 & a(2)==12
    c(3)=b(3)-a(3)+31;
  end

  b(2)=b(2)-1;
end

%month
if b(2)>= a(2)
  c(2)=b(2)-a(2);
else
  c(2)=b(2)-a(2)+12;
  b(1)=b(1)-1;
end

%years
c(1)=b(1)-a(1);
```

```
c;
%The amount of total minutes
c_minutes=c(3)*24*60+c(4)*60+c(5);



diff_t=c;

diff_minutes=c_minutes;

$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
function f=a3_max(a)
 switch (a(2))
 case 1
    f=31;
 case 3
    f=31;
 case 5
    f=31;
 case 7
    f=31;
 case 8
    f=31;
 case 10
    f=31;
 case 12
    f=31;

 case 4
    f=30;
 case 6
    f=30;
 case 9
    f=30;
 case 11
    f=30;

 otherwise %a(2)=2
```

```
   if mod(a(1),4)<0.001
     f=29;
   else
     f=28;
   end
 end
 $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
```

Block2:

```
% this prigram is to generate pilots
Total_pilots=54,
Total_copilots=58,

for i=1:(Total_pilots + Total_copilots)%******************for
pilot(i).ID=i;%tracking Number
pilot(i).name=i;%name

if i<=Total_pilots
pilot(i).rank=1;%1 is pilot,2 is co-pilot
else
pilot(i).rank=2;%1 is pilot,2 is co-pilot
end

pilot(i).status=0;  %0 is waiting,1 is flying
pilot(i).starttime=[0 0 0 0 0 0];  %a start time of a task
pilot(i).endtime=[0 0 0 0 0 0];  %a end time of a task
pilot(i).flyingtime=0;  %a flying time of a task

if i<=Total_pilots
  pilot(i).accumulatetime=1000;  %the accumulated flying time
else
  pilot(i).accumulatetime=600;  %the accumulated flying time

end

end%************************************end for**********


i=i+1;
```

```
%%%%%%%the ending flag*****************************************
pilot(i).ID=99999;
pilot(i).name='papa';
pilot(i).rank=0;
pilot(i).status=0;
pilot(i).starttime=0;
pilot(i).endtime=0;
pilot(i).flyingtime=0;
pilot(i).accumulatetime=0;
i,

$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
%*******update_pilots nformation

load('pilotsave');

Na=54;%the number of first pilots
Nb=58;%the number of co-pilots
Np=Na+Nb;%the number of all pilots

for i=1:(Np+1)
   pilot(i).nomatch=0;
end

%***set the information as to nomatch

pilot(3).nomatch=[Na+1 Na+7 Na+15 0];
pilot(9).nomatch=[Na+1 Na+13 Na+19 Na+22 0];
pilot(33).nomatch=[Na+14 Na+17 Na+29 0];
pilot(41).nomatch=[Na+9 0];
pilot(52).nomatch=[Na+14 Na+17 0];

save pilotsave pilot;

$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

% initialize_rostering
pilot_generate;
generate_task;

save pilotsave pilot;
```

```
save tasksave task;

f_p_d=17;%the number of tasks per day

Na=54;%the number of first pilots
Nb=58;%the number of co-pilots
Np=Na+Nb;%the number of all pilots

$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
%set_initial_parameter


f_p_d=17;%the number of tasks per day

Na=54;%the number of first pilots
Nb=58;%the number of co-pilots
Np=Na+Nb;%the number of all pilots

load('tasksave')
task_pointer=1
time_control=task(1).starttime
d_x=1

$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
function f=sort_pilot(a)
n1=size(a);
n=n1(2);

for i=1:n
  for j=i+1:n
    if a(j).accumulatetime < a(i).accumulatetime
      box_a=a(i);
      a(i)=a(j);
      a(j)=box_a;
    end%end if
  end%end for
end%end for

f=a;

$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
```

```
%sort tasks in the decsending order according to the flying times
function f=sort_task(b)
n1=size(b);
n=n1(2);

for i=1:n
  for j=i+1:n
    if b(j).flyingtime > b(i).flyingtime
      box_b=b(i);
      b(i)=b(j);
      b(j)=box_b;
    end%end if
  end%end for
end%end for

f=b;

$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
% this program is to is to find the shortest path
% between the start node and the ending node
function [path,length]=shortest_path(w)

n=size(w);
n=n(1);

%**initialize**********

k=1;

u(1,1)=0;R(1)=1;

for j=2:n
  u(j,1)=w(1,j);
  R(j)=1;
end

%***********

k=k+1;

u(1,2)=0;
```

```
for j=2:n
  x_min=u(j,1);
  for i=2:n
     if x_min > (u(i,1)+w(i,j))
        x_min = (u(i,1)+w(i,j));
        R(j) = i;
     end
  end
  u(j,2)=x_min;
end

%**end initialize**********

while( norm(u(:,2) - u(:,1),2)>(1.0e-4) ) & (k < n)%***while
   k=k+1;

   u(:,1)=u(:,2);

for j=2:n
  x_min=u(j,1);
  for i=2:n
     if x_min > (u(i,1)+w(i,j))
        x_min = (u(i,1)+w(i,j));
        R(j) = i;
     end
  end
  u(j,2)=x_min;
end

end%**************************************************end while

%u=u';

k=k-1;
length=u(n,1);
%R;

%***We will find the shortest path between node 1 and node n
j=1;
p(j)=n;
```

```
j=j+1;
p(j)=R(n);

while(abs(p(j)-1)>1.0e-4)
  j=j+1;
  p(j)=R(p(j-1));
  end

nk=j;
%p;
%**************************************find the final result
j=1;
while (j <= nk/2 )
  x=p(j);
  p(j)=p(nk-j+1);
  p(nk-j+1)=x;
j=j+1;
end
%p,

path=p;%display the shortest path between node 1 and node n
u=[];%*******************release the space that is occupied

$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
%f_pair_finding
%This program is to find the min cost matching
function f=f_pair_finding(VU,RR)%the first is the input ajacent matrix
                      %the second is the needed flow volume



Inf_B=1.0e7;
%m1%pilots selected
%m2%copilots selected
[m1 m2]=size(VU);


%RR%needed flow volume

%****************************initialize the cost matrix
FEE=Inf_B*ones(m1+m2+2,m1+m2+2);
```

```
for i=1:(m1+m2+2)%diagno
  FEE(i,i)=0;
end
%******************
for j=2:(m1+1)
  FEE(1,j)=0;
end

for i=2:(m1+1)
  FEE(i,1)=Inf_B;
end

for j=(m1+2):(m1+m2+1)
  FEE(m1+m2+2,j)=Inf_B;
end

for i=(m1+2):(m1+m2+1)
  FEE(i,m1+m2+2)=0;
end
%*******************

%***************set the main weight****
%VU;
UV=Inf_B*zeros(m2,m1);

FEE(2:m1+1,m1+2:m1+m2+1)=VU;
FEE(m1+2:m1+m2+1,2:m1+1)=UV;
%****end of setting the mail weight****

%*********add fine weight************

%*********end of adding fine weight**


%*******************end of initializing the cost matrix

%'initial FEE'
%FEE,

%***************************initialize the flow matrix
```

```
FL=zeros(m1+m2+2,m1+m2+2);
%******************end of initializing the flow matrix

FLV=sum(FL(1,2:m1+1));%initialize the flow volumn

while (FLV < RR)%************************while********
  path=shortest_path(FEE);%find the shortest path from
                    %the cost matrix
  np=size(path);
  np=np(2);%how many nodes along this shortest path

  %%1%%%%update the flow matrix************************
  k=1;
  while(k < np)%****************while
    k=k+1;
    if path(k-1) < path(k)%'+arc'
      FL(path(k-1),path(k))=FL(path(k-1),path(k))+1;
    end

    if path(k-1) > path(k)%'-arc'
      FL(path(k),path(k-1))=FL(path(k),path(k-1))-1;
    end

  end%**********************end while
  %%%%%%end of updating the flow matrix**************
  %'FL'
  %FL,

  FLV=sum(FL(1,2:m1+1));%compute the flow volumn

  if abs(FLV - RR)<1.0e-4
    break;
  end

  %%2%%%%update the cost matrix************************
  k=1;
  while(k < np)%****************while
    k=k+1;

      FEE(path(k),path(k-1))=-FEE(path(k-1),path(k));
```

```
        FEE(path(k-1),path(k))=Inf_B;

    end%**********************end while
    %%%%%%end of updating the cost matrix**************
    %'FEE'
    %FEE,

end%************************************end while****

pc=FL(2:m1+1,m1+2:m1+m2+1);%the pairs matrix we get
%pc,

%%%************************find the pairs***********
k=1;
for i=1:m1
    for j=1:m2

        if abs(pc(i,j)-1)<1.0e-4
            pair(:,k)=[i;j];
            k=k+1;
        end

    end
end
%pair,%the pairs we finally get
f=pair;
%%%********************end of finding the pairs******
```

$$

Block3: The main program

```
%rostering
%  this is the main program for rostering problem using sort method

load('tasksave');%open task table

%task_pointer=1;%set the pointer that points the task

NN1=size(task);
```

```
NN=NN1(2);%the number of tasks

%f_p_d=17;%the number of tasks per day

%time_control=task(1).starttime;%the control variable of time axes

%while task_pointer < NN %**********************main circle while

 load('pilotsave');%open pilot table
 %Na=54;%the number of first pilots
 %Nb=58;%the number of co-pilots
 %Np=Na+Nb;%the number of all pilots

 %the following block1 is to set the status paramerters of the pilots******1
 for i=1:Np

   d_time_control=datenum(time_control(1),time_control(2),time_control(3),...
     time_control(4),time_control(5),time_control(6));

   start_time=pilot(i).starttime;
   d_start_time=datenum(start_time(1),start_time(2),start_time(3),...
                  start_time(4),start_time(5),start_time(6));

   end_time=pilot(i).endtime;
   d_end_time=datenum(end_time(1),end_time(2),end_time(3),...
                  end_time(4),end_time(5),end_time(6));


   if (d_time_control - d_start_time) >= (d_end_time - d_start_time)
     pilot(i).status=0;
     %pilot(i).starttime=0;
     %pilot(i).endtime=0;
   end %end if
end %end for
%********end of block1*****************************************************1


%The follwoing block2.1 is to select pilots for assigning tasks***********2.1

 %**1**select first pilots
 j=1;
```

```
for i=1:Na
  if abs(pilot(i).status) < 0.001
    a_pilot(j)=pilot(i);
    j=j+1;
  end%end if
end%end for
m1=j-1;
%**2**select co-pilots
j=1;
for i=(Na+1):(Na+Nb)
  if abs(pilot(i).status) < 0.001
    b_pilot(j)=pilot(i);
    j=j+1;
  end%end if
end%end for
m2=j-1;
%sort in ascending order according to accumulated flying time

a_pilot=sort_pilot(a_pilot);
b_pilot=sort_pilot(b_pilot);

%%************************************************************
%%**we will solve a weight matching problem******************

%%%%block 1:generate the matrix VU%%%%%%%%%%
A=zeros(m1,m2);%the matrix VU
small=0;
if (m1 < m2)%****************************if (m1 < m2)
  for i=1:m1
    for j=1:m2
      tu=b_pilot(j).accumulatetime;
      tu0=b_pilot(i).accumulatetime;
      A(i,j)=abs(tu-tu0);
    end
  end
%****adjust the set-up of the weight
for i=1:m1

  if (i-1)<1.0e-4
    A(i,i)=small*i;
    A(i,i+1)=small*i;
```

```
  else
    A(i,i)=small*i;
    A(i,i-1)=small*i;
    A(i,i+1)=small*i;
  end

end
%****adjust the set-up of the weight
end%**********************************end if (m1 < m2)

if abs(m1 - m2)<1.0e-4%****************************if (m1 = m2)
  for i=1:m1
    for j=1:m2
      tu=b_pilot(j).accumulatetime;
      tu0=b_pilot(i).accumulatetime;
      A(i,j)=abs(tu-tu0);
    end
  end
%****adjust the set-up of the weight
for i=1:m1

  if (i-1)<1.0e-4
    A(i,i)=small*i;
    A(i,i+1)=small*i;

  elseif (m1-i)<1.0e-4
    A(i,i)=small*i;
    A(i,i-1)=small*i;
  else
    A(i,i)=small*i;
    A(i,i-1)=small*i;
    A(i,i+1)=small*i;
  end

end
%****adjust the set-up of the weight
end%**********************************end if (m1 = m2)

if (m1 > m2)%*****************************if (m1 > m2)
  for j=1:m2
```

```
    for i=1:m1
      tv=a_pilot(i).accumulatetime;
      tv0=a_pilot(j).accumulatetime;
      A(i,j)=abs(tv-tv0);
    end
  end

  %****adjust the set-up of the weight
  for j=1:m2

  if (j-1)<1.0e-4
    A(j,j)=small*j;
    A(j+1,j)=small*j;

  else
    A(j,j)=small*j;
    A(j-1,j)=small*j;
    A(j+1,j)=small*j;
  end

end
%****adjust the set-up of the weight

end%*********************************end if (m1 > m2)


INF_B=1.0e7;
for i=1:m1%*********************************************for
  if norm(a_pilot(i).nomatch,2) > 1.0e-4%**********if
    nomat=a_pilot(i).nomatch;
    for j=1:m2%*********************************for
      k=1;
      while(nomat(k) > 1.0e-4)%**********while
        if abs(nomat(k) - b_pilot(j).ID) < 1.0e-4
          A(i,j)=INF_B;
        end
        k=k+1;
      end%**************************end while
    end%************************************end for
  end%*******************************************end if
end%************************************************end for
```

```
pair=f_pair_finding(A,f_p_d);%*******************************find the pairs

 for k=1:f_p_d
   a_pilot_select(k)=a_pilot(pair(1,k));
   b_pilot_select(k)=b_pilot(pair(2,k));
 end

 %%**end of solving a weight matching problem*****************
 %%***********************************************************

 a_pilot=a_pilot_select;%select those first pilots for assignment
 b_pilot=b_pilot_select;%select those co-pilots for assignment

 %end of block2.1 which is to select pilots for assigning tasks************2.1


%The following block2.2 is to select tasks for taking*********************2.2

 j=1;
 for i=task_pointer : (task_pointer + f_p_d - 1)
   f_task(j)=task(i);
   j=j+1;
 end %end for

 %sort in descending order according to flying time
 f_task=sort_task(f_task);

%end of block2.2 which is to select tasks for taking**********************2.2


%The following block3 is assigning process*********************************3
for i=1:f_p_d
  a_pilot(i).starttime=f_task(i).starttime;
  b_pilot(i).starttime=f_task(i).starttime;

  a_pilot(i).endtime=f_task(i).endtime;
  b_pilot(i).endtime=f_task(i).endtime;

  a_pilot(i).flyingtime=f_task(i).flyingtime;
  b_pilot(i).flyingtime=f_task(i).flyingtime;
```

```
  %%increase the accumulated flying time**********************************
  a_pilot(i).accumulatetime=f_task(i).flyingtime + a_pilot(i).accumulatetime;
  b_pilot(i).accumulatetime=f_task(i).flyingtime + b_pilot(i).accumulatetime;

  a_pilot(i).status=1;
  b_pilot(i).status=1;


  f_task(i).pilot=a_pilot(i).name;
  f_task(i).copilot=b_pilot(i).name;
end %end for

%End of block3 which is assigning process*********************************3

%The following block4 is to record the assignment************************4

 %**1**record for first pilots

 for i=1:Na
  for j=1:f_p_d

    if abs(pilot(i).ID - a_pilot(j).ID) < 0.001
      pilot(i)=a_pilot(j);
    end%end if

  end %end for
 end%end for

 %**2**record for co-pilots
 for i=(Na+1):(Na+Nb)
  for j=1:f_p_d

    if abs(pilot(i).ID - b_pilot(j).ID) <0.001
      pilot(i)=b_pilot(j);
    end%end if

  end %end for
 end%end for

 %**3**record for tasks
```

```
for i=task_pointer : (task_pointer + f_p_d - 1)
   for j=1:f_p_d

    if abs(task(i).number - f_task(j).number) <0.001

    task(i)=f_task(j);

    end%end if

  end% end for
 end %end for

%**4**update the pilot table and task table
save pilotsave pilot;
save tasksave task;
%End of block4 which is to record the assignment*************************4

%%%%%%%%%%
%   1-4  %
%%%%%%%%%%

 %%******display some of the results************************************5

   for i=1:Na
      xa(i)=pilot(i).accumulatetime;
   end

   j=1;
   for i=(Na+1):(Na+Nb)
      xb(j)=pilot(i).accumulatetime;
      j=j+1;
   end

   std_pilot=std(xa);
   std_copilot=std(xb);

   xax=1:1:Na;
   xbx=1:1:Nb;

   %d_x=time_control(3);
```

```
    subplot(2,2,1),stem(xax,xa,'.b'),
    subplot(2,2,2),plot(d_x,std_pilot,'.b'),hold on
    subplot(2,2,3),stem(xbx,xb,'.r'),
    subplot(2,2,4),plot(d_x,std_copilot,'.r'),hold on

    s1(d_x)=std_pilot;
    s2(d_x)=std_copilot;

  %%******end of display some of the results*****************************5

  %************increase the pointer variable*******************main pointer
  task_pointer = task_pointer + f_p_d;
  time_control(3)=time_control(3)+1;
  d_x=d_x + 1;
  %************end increase the pointer variable***************main pointer

  if task_pointer > NN
    task_pointer = 1;
    next_cycle=task(1).starttime;
    next_cycle(2)= next_cycle(2)+1;

    if next_cycle(2) > 12
      next_cycle(2)=1;
      next_cycle(1)=next_cycle(1) + 1;
    end

    task(1).starttime = next_cycle;
    time_control=task(1).starttime;
  end

%end %*****************************************end of main circle while
```

REFERENCES

[1] Ackoff RL (1977). Optimization + objectivity = opt out. European Journal ofOperational Research 1: 1-7.

[2] Andersson, E., Housos, E., Kohl, N., Wedelin, D.: Crew pairing optimization, in: OR in airline industry, Eds. Yu, G., Kluwer Acad. Publ., Boston, 1997.

[3] Arabeyre, J.P., Fearnley, J., Steiger, F.C., Teather, W., The airline crew scheduling problem: a survey, Transport. Sci. 3, 1969, 140-163

[4] Baker, E.K., Bodin, L.D., Finnegan, W.F., Ponder, R.J., Efficient heuristic solutions to an airline crew scheduling problem, AIIE Trans. 11, 1979, 79-85.

[5] Barnhart, C., Johnson, E. L., Nemhauser, G. L., Savelsbergh, M. W. P., Vance, P. H., 1998. Branchand-Price: column generation for solving huge integer programs, Operations Research 46 316-329.

[6] Bianco, L., Bielli, M., Mingozzi, A., Ricciardelli, S., Spandoni, M., A heuristic procedure for the crew rostering problem, Europ. J. Oper. Res. 58, 1992, 272-283.

[7] Buhr, J. (1978). Four Methods for Monthly Crew Assignment-acomparison 0f Efficiency.1978 AGIFORS Synposium Proceedings, 18,403-430.

[8] Caprara, A., Fischetti, M., Toth, P., Vigo, D.: Modeling and solving the crew rostering problem, Technical report DEIS-OR-95-6(R), University of Bologna, 1995; to appear in Operations Research.

[9] Cattrysse, D.G., Van Wassenhove, L.N., A survey of algorithms for the generalized assignment problem, Europ. J. Oper. Res. 60, 1992, 260-272.

[10] Clayson J (1984). Micro-Operational Research: A Simple Modeling Tool for Managers.In: J. Richardson (ed). Models of Reality, Lomond: Mt.Airy, Maryland, Chapter 16.

[11] Desaulniers, G., Desrosiers, J., Gamache, M., Soumis, F., 1998. Crew scheduling in air transportation. In: Crainic, T.G., Laporte, G. (Eds.), Fleet Management and Logistics, Kluwer Acadamic Publishers, Boston, MA, pp. 169-185.

[12] Desaulniers, G., Desrosiers, J., Ioachim, I., Solomon, M.M., Soumis, F. A unified framework for deterministic time constrained vehicle routing and crew scheduling problems, Technical report G-94-46, GERAD Montréal, December 1994.

[13] Desaulniers, G., Desrosiers, J., Dumas, Y., Marc, S., Rioux, B., Solomon, M.M., Soumis, F. Crew pairing at Air France, Europ. J. Opl. Res. 97, 1997, 245-259.

[14] Desaulniers, G., Desrosiers, J., Dumas, Y., Marc, S., Rioux, B., Solomon, M.M., Soumis, F. (1993). Crew Pairing at Air France. Les Cahiers du GdRAD, G-93-89, Ecolc des Hauies Efvdes, Commcrciales, Alonfrial, Canada, H3T lV6. Revision Mai 1995.

[15] Desaulniers, G., Desrosiers, J., Ioachim, I., Solomon, M.M., and Soumis, F. (1994). A Unified Framework for Deterministic Time 1 Constrained Vehicle Routing and Crew Scheduling Problems. les Cahiers 1 du G8RAD, G-94-46, Ecole des Hautes Efudes Commerciales, Montrieal, 1 Canada, H3T 1V6.

[16] Desrochers, M., Gilbert, J., Sauve, M. and Soumis, F. (1990). Crew-OPT: Subproblem Modeling in a Column Generation Approach to Urban Crew Scheduling. Les Cakiers du GERAD G-94-46, Ecole des Hautes Etudes Commerciales, Montreal, Canada, H3T l V6.

[17] D.S.Qian. Operation Research. Beijing, Tshinghua, 2000.

[18] Foulds L R (1983). The heuristic problem-solving approach. Journal of the Operational Research Society 34**:** 927-934.

[19] Glanert, W. (1984). A Timetable Approaeh to the Assignment of Pilots to Rotations. 1984 AGIFORS Symposium Proceedings, 24, 369-391.

[20] Gershkoff, I., Optimizing flight crew schedules, Interfaces 19 (4), July-August 1989, 29-43.

[21] Graves, G., McBride, R., Gershkoff, I., Anderson, D., Mahidhara, D., Flight crew scheduling, Management Science 39, 1993, 736-745.

[22] Ignizio JP (1980). Solving large-scale problems: a venture into a new dimension. Journal of the Operational Research Society **31**: 217-225.

[23] Jiang P., Lei Ji (2000). Operation research in Air China, Chinese Journal of Management Science: V6 PP 213-225.

[24] Lenat DB (1982). The nature of heuristics. Artificial Intelligence **19**: 189-249.

[25] Marchetfini, F. (1980). Automatic monthly cabin crew rostering procedure 1980, AGIFORS Symposium Proceeding, 20, 23-59.

[26] Michalewicz Z and Fogel DB (2000). How to Solve It: Modern Heuristics Springer-Verlag: Berlin.

[27] Morton TE and Pentico DW (1993). Heuristic Scheduling Systems. Wiley – Interscience: New York.

[28] Müller-Mehrbach H (1981). Heuristics and their design: a survey. European Journal of Operational Research **8**:1-23.

[29] Muller-Malek H, Matthys D, and Nelis E (1997). Heuristics and expert- like systems. Belgian Journal of Operations Research, Statistics, and Computer Science **27:** 25-63.

[30] Nicoletti, B. (1975). Automatic Crew Rostering.Transpotation Science, 9, 33-42.

[31] Papadimitriou C H, Steiglitz K. Combinatorial Optimization: Algorithm and Complexity. New Jersey: Prentice-Hall INC, 1982.

[32] Pinedo M and Simchi-Levi D (1996). Heuristic methods. In: M. Avreil and B. Golany (eds.). Mathematical Programming for Industrial Engineers. Marcel Dekker: New York, pp. 575-617.

[33] Reeves CR (ed) (1993). Modern Heuristic Techniques for Combinatorial Problems. Halsted Press: New York.

[34] Ryan, D.M. and Palkner, J.C (1988). On the Integer Properties of Scheduling Set Partitioning Models. European Journal of Operatioal Research, 35, 442-456.

[35] Ryan, D.M. (1992). The Solution of Masive Generalized Set Partitioning Problems in Air Crew Rostering. Jounal of the Operational Research Society, 43, 459-467.

[36] Ryan, D.M. Optimization Earns Its Wings: The development of crew scheduling systems for Air New Zealand. OR/MS 4, 2000.

[37] Reeves C R (Ed). Modern Heuristic Techniques for Combinatorial Problems. Oxford: Blackwell Scientific Publications, 1993.

[38] Sanso, B., Desrochers, M., Desrosiers, J., Dumas, Y. And Soumisi F. (1990) Modeling and Solving Routing and Scheduling Problems: GENCOL User Guide. GENAD, Ecole des Haufes Etudes.Comm'cr-1 cialesv AloRirial, Quibec, H3T l V6.

[39] Tingley, G.A. (1979). Still Another Solution Method for the Monthlv Aircrew Assignment Problem, 1979 AGIFORS Symposium Proceedings, 19, 143-203.

[40] W.X. Xing, J.X.Xie. Modern methods in optimization, Beijing, Tshinghua, 2000.

[41] W.X. Xing, J.X.Xie. Network optimization, Beijing, Tshinghua, 2000.

[42] Yu, G. Recent Advances in Optimization Applications in the Airline Industry. Guest Editor for the Special Issue of the Journal of Combinatorial Optimization, 1, 3, 1997.

[43] Yu, G. (ed.). Operations Research in the Airline Industry. Kluwer Academic Publishers, Boston, MA, total 460 pages, 1997.

[44] Yu, G. and B. Thengvall."Optimization in the Airline Industry." Handbook on Applied Optimization, edited by P.M. Pardalos and M.G.C. Resende, Oxford University Press, 2001.

[45] Yu, G. and B. Thengvall. "Airline Optimization." Encyclopedia for Optimization, edited by D.Z. Du, Kluwer Academic Publishers, 2000.

[46] Zanakis SH, Evans JR, and Vazacopoulos, AA (1989). Heuristic methods and applications: a categorized survey. European Journal of Operations Research 43: 88-110.

BIOGRAPHICAL SKETCH

Xugang Ye is currently a graduate student in Applied Mathematics, Department of Mathematics of Florida State University. Before he came to Florida State University, he received a B.S. in Thermo Physics & Engineering, with specialization on Fire Science & Engineering, from Department of Thermo Science & Energy Engineering, University of Science & Technology of China (USTC), Hefei, Anhui, P.R.C. He also received his M.S. in Applied Optimization and Operation Research from Institute of Policy and Management Science, Chinese Academy of Sciences (CAS), Beijing, P.R.C. His current research interests include: Large Scale Numerical Optimization, Combinartorial optimization & Computational Complexity, Computational methods in Statistics, and Computational Dynamics.