ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Aircraft Scheduling

Master's Thesis by

Keller Peter and Schilling Simon

MA-2004-10, Summer Term 2004
October 15th, 2004

Advisor: Dr. Alexander Hall
Supervisor: Prof. Dr. Thomas Erlebach

Abstract

This thesis presents analyses of algorithms for aircraft scheduling. The main focus applies to tail assignment. Local search, tabu search, simulated annealing, and modifications of these technics are used for the optimization. Simulated annealing based algorithms outmatches the others because of the topology of the solution space with its huge number of local minima.

Contents

Chapter 1

Introduction

Today's big airlines could save several million dollars by reducing the costs of each takeoff by only 50 dollars. This example illustrates how important it is to optimize the processes and planning that take place before an aircraft can take off. [1]

Already two years before takeoff, planning starts by estimations of passenger numbers. Based on these numbers, decisions are made about which cities to connect and the frequency of these flights. About one and a half years before takeoff, the fleets are assigned to the flights. After that, a plan for each aircraft is made, accounting also for the needed maintenances — this is the focus of this thesis. Six weeks before takeoff, the crews are assembled. Notice that one can improve the contentment of the crews by accounting for personal preferences like flight destinations or personnel who likes to work together. A content crew will increase the quality of its service, therefore also increasing the contentment of the passengers. But even if everything up to this point is done perfectly, unexpected events like illness or technical problems will force the airlines to change their plans. One could therefore judge the quality of a flight schedule not only by the costs, but also by the additional cost that unexpected events could introduce. Actually the term "unexpected events" is then no longer valid, because we expect them to happen.

One can see that the process of planning is split into many phases that are handled independently of their preceding phase. This reduces the complexity, but also introduces the possibility of "bad" decisions in an early stage that will not be changed later. Thus, there are also efforts of merging these stages again.

It is important to see that it is not only important how *good* a solution to all these problems is, but also how *fast* it can be found. Faster solutions allow later planning, therefore the airline is more flexible. And, at least for unexpected last minute events, a fast reaction is essential.

Most of these problems are NP-hard, this means there is no known algorithm (it is assumed that there *is* no algorithm) which is able to find the optimum of a real-size problem in feasible time. NP-hardness is common for transportation problems, a famous one is for example the travelling salesman problem. So, the target is not to find the optimal solution, but to get a good trade-off between quality and calculation time.

Chapter 2

Problem Description

People who deal with flight schedule optimization have generally two missions. Before any aircraft can lift off to transport tourists and businessmen to their destination, it has to be decided which aircraft should perform which flight. This problem is called *tail assignment*. Once an aircraft and a crew are allocated to every flight, the turbines can be powered-on, if it were not for Murphy's law. Aircrafts break down, pilots get sick or a storm prevents take offs. To manage these problems *schedule recovery* is used. We will be discussing these two topics in the next two sections.

2.1 Tail Assignment

One and a half years before take off, the allocating of aircraft types to the flights takes place. Here it is for instance made sure that a domestic flight will not be flown by a Boeing 747 as well as that an intercontinental flight will never be performed by an Airbus A319. This allocation is called *fleet assignment*. The next step is to allocate aircrafts to the flights, doing the tail assignment.

The main difficulty is not to breach any of the numerous constraints. There are basically two types of constrains: hard and soft ones. Hard constraints are given by nature or by law. An aircraft has to depart from the same airport at which it has arrived before. It cannot depart earlier than its arrival time. And of course it is not possible that one half of an airplane performs one flight while the other half flies to another airport. That every flight has to be performed and every airplane has its own maintenance days which are not interchangeable with other airplanes, are other hard constraints. Soft constraints can be broken but it is not beneficial for the purse. There are limits for the number of flights and flight hours between two maintenance events. Passengers should be conveyed and not left at the airport. The seating of an airplane can be changed, for example by replacing business class seats with economy class seats, however these changes are not free. Every time a soft constraint is violated the total costs will rise. A complete description of the costs is given in Sec. 5.1.2. The target is to minimize these costs.

2.2 Schedule Recovery

There are plenty of more or less little things that put an airline into a queasy situation. Thunderstorms avert landings and take offs, black ice can effect the same. An aircraft can break. A pilot might suffer from a food poisoning and is

thus not able to fly the plane. These and many more incidents can mix up the flight schedule. The effort to try to readjust everything is called *schedule recovery*. There are a couple of possibilities for such a recovery. If an unusable airport is the matter of fact the solution can be the use of another airport, and then trains and buses to take the people to their final destination. Sometimes it is much more difficult to decide what to do. In a case where an airplane is late or broken it is not easy to say what is the best back door. Is it better to use a backup aircraft, which might result in delays for a lot of flights or should one flight be cancelled without affecting others? These and other questions have to be answered quickly. Mostly, it is hard to say what effect a change will have. Theory and reality can differ highly. The real costs with or without changes are hard to evaluate. The reasons are raised costs for side effects like changed working hours for the employees or logistic problems.

Chapter 3

Preliminaries

This chapter gives explanations about general structures, terms, and definitions which are used by the algorithms presented in Chap. 4. We used data which we generated on our own because the ones we got from Lufthansa are defective, see also Appendix A.

3.1 Definitions

A *leg* is nothing more than a flight performed by an aircraft. A leg is described by its parameters. They are *leg type, departure airport, arrival airport, departure time, arrival time, leg state, and number of passengers (first, second, and economy class)*. The leg type defines if it is a normal flight or a maintenance event. The leg state informs if it is already arrived, departed, cancelled, etc. The others are self-explanatory. A group of connected legs which are consolidated are called *cluster*. These legs must be operated by the same aircraft. In our problem instances all clusters contain only one leg so that these two terms are interchangeable.

3.2 Structures

It is necessary to use a suitable structure to make the implementation of the algorithms as easy as possible. In our case a special *directed graph* is an adequate structure. A directed graph $G = (V, E)$ is a set of vertices $V = \{v_1, \ldots, v_n\}$ and edges connecting these vertices $E = \{e_1, \ldots, e_m\}$ where $e = (v, w)$ is an edge between vertex v and vertex w, pointing at w. A graph can be used as a descriptive way to show the relations between the clusters. We decided to use the same representation as the group from Lufthansa Systems is using. The clusters are represented by the vertices. There is an edge between vertex A and vertex B if it is possible to perform cluster B directly after cluster A. This means cluster A's arrival airport is the same as cluster B's departure airport and cluster A's arrival time is earlier then cluster B's departure time. To reduce the number of edges of the graph and thus the size of the problem it is reasonable to use a barrier concerning the time between arrival and departure. It does not make any sense to connect two clusters by an edge if the time between arrival of the first and departure of the second cluster is a month. What is a reasonable barrier? If it is too short, valid solutions get lost, if it is too long, the size increases massively. A good choice seems to put it somewhere between three and seven days.

A closer look at the graph concerning the aircrafts shows that not every cluster can be flown by every aircraft. This is because of the maintenance events. These

events are fixed in time and are for a specific aircraft. I.e. each aircraft regularly has to be at certain predefined airports to get maintained. Thus, this aircraft is not able to perform flights during these times. Therefore, it is reasonable to consider only the flights an aircraft is actually able to fly. We used an attempt proposed by Barnhart et al. [2]. The goal is to find *strings*. A string is the group of flights an aircraft performs between two maintenance events. If we examine each string by itself and bring all clusters together that can be in a certain string we get *subgraphs*. A subgraph has a maintenance event as the first vertex and the following maintenance event as the last vertex. It contains all vertices of the whole graph that can be reached from the first maintenance event **and** from which the end maintenance event can be reached. The goal is to find in each subgraph exactly one path from the start maintenance to the end maintenance vertex. The difficulty is that most clusters are part of more than one subgraph, but every cluster can be flown only by one aircraft, i.e. every cluster can be only contained by one string.

3.3 Using an Overflow Aircraft

An initial solution does not necessarily need to be already a good solution. One could even define that in the initial solution, the hard constraints do not have to be met (e.g. not all legs must be flown). In the model, such unmet constraints can be realized by *overflow aircrafts*, which perform the legs that are not flown. In this case, the optimizer must take care of finding a solution which meets all conditions, so it will penalize unmet conditions (legs flown by overflow aircrafts) heavily. The advantage of this approach is that it is much easier to find an initial solution, because it need not meet the hard constraints. On the other hand, this leads to an optimizer which also needs to deal with unmet hard constraints.

The advantage of an initial solution where all hard constraints are met, is a less complex optimizer, which will always stay in the solution space limited by the hard constraints. The risk of this approach is, that it might be easier for the optimizer to find a good solution, when it also can take some steps over the “forbidden” solutions, instead of always staying in the “allowed” solution space. Think of it as a “shortcut”. To overcome this, it is necessary to ensure that the solution space is well connected, so that such shortcuts are not needed.

Chapter 4

Initial Solution

This chapter elucidates the different algorithms we propose to find a basic solution. The results of an extensive empirical evaluation are given in Chap. 6. Such solutions should not violate any hard constraints. They will be used as starting points for the optimizer algorithms explained in Chap. 5.

The reader may wonder why we gave such unordinary names to the algorithms. There is just one simple reason. We did not find any meaningful names which are shorter then a whole sentence. And just using our own name — like the most algorithms are labelled by their developer — is in our opinion uncool, so we used the names of movies.

4.1 A Correct Solution

Before we can start searching a solution, we have to define what a solution is. A correct solution allocates the clusters to the aircrafts in a way, that it is possible for all the aircrafts to perform these allocated clusters without violating any hard constraints. But that is not enough. It is demanded that all clusters are allocated.

4.2 The First Attempt

The assumption that an initial solution can be found quite easily brought us to the idea of a straightforward algorithm. Considering subgraphs one by one is the basic idea. The subgraphs will be processed in order of the departure time of its first maintenance cluster, or the arrival time of its last maintenance cluster. These two possibilities yield the same result. That is because the time span of all the subgraphs (time between the departure of the first and the arrival of the last maintenance cluster) is almost the same and therefore the order is more or less equal. As soon as a cluster is assigned to a subgraph it will be cancelled in all the other subgraphs and in the graph. A cluster will only be assigned if it is a successor of the last which was assigned. There are two possibilities for backtracking. When cancelling a cluster in a subgraph, because it was assigned to another, it is possible that there is no longer a connection between the start and the end maintenance cluster. So it is necessary to backtrack. The other reason for backtracking is the isolation of a cluster in the graph. I.e. that a cluster can lose all outgoing edges because all succeeding clusters have been cancelled. This can happen without that any subgraph loses the connection between start and end cluster.

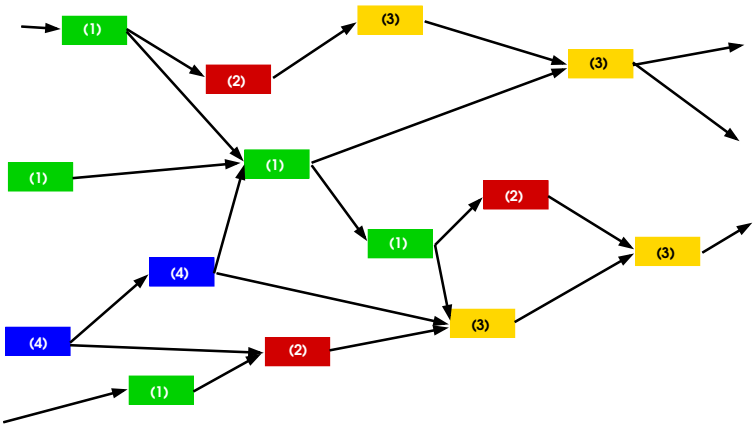

Figure 4.1: The end cluster problem.

We first thought that this algorithm works well, but then we had a closer look at the computed assignments. Much to our surprise we found that there are clusters that have not been assigned. The reason was found in the clusters at the very end of the graph. Have a look at Fig. 4.1, which shows the last part of the graph. The red (2) clusters are maintenance clusters, the greens (1) allocated clusters. There is no path from a yellow (3) cluster to a maintenance cluster, therefore it is not possible to allocate them, but that is no problem. The blue (4) ones are clusters with a path to a maintenance cluster, but which are not allocated. The algorithm is not able to prevent this, because it only backtracks when a not allocated cluster loses all outgoing edges. The blue (4) ones will never lose the path to a yellow (3) cluster and so never lose all outgoing edges.

The First Attempt

1. Take the subgraph with the earliest departure time that has not been considered yet.
2. Assign a cluster to the subgraph which has a common edge with the last assigned cluster.
3. Delete this cluster in all the other subgraphs (if it is contained).
4. If a subgraph lost the connection between its two maintenance clusters during step 3 → backtracking.
5. Go on with step 2 until the last inserted cluster has a connection to the second maintenance cluster of the subgraph and it is not possible to put another cluster between these two.
6. Go on with step 1 until all subgraphs are performed.

There was no way to prevent this fault without slowing down the algorithm immensely. One attempt is after finishing a subgraph to check if there is still a cluster not allocated whose arrival time is earlier then the departure time of the first maintenance cluster of the last performed subgraph. If there is one, it is necessary to backtrack. The loophole was an algorithm with another approach. So Shrek saw the light of day.

4.3 Shrek

We found that it is smarter to assign the legs chronologically to the subgraphs, which means the legs are sorted according to their departure time, and then assigned one by one to an aircraft[1] which can fly this leg.

Shrek

1. Get the first/next leg l, sorted by departure time.
2. Search a valid aircraft for leg l (the first/next possible aircraft). If there are no more aircrafts, return *no success* (backtrack)
3. If l is the last leg, a solution is found
4. Recursively call (1)
   (a) When *no success* is returned, try another aircraft (Step 2)

4.3.1 Choosing an Aircraft

We used two different methods of choosing an aircraft in this algorithm. One possibility is to choose aircraft 1 first. If that does not work, aircraft 2 etc. If none of the aircrafts leads to a solution, backtracking is needed. The second possibility is choosing an aircraft randomly. In this case, the algorithm behaves differently every time it runs. This is not really a big help for finding a solution, but helped us getting a better insight to the nature of the whole problem, because this way it can be distinguished between problems that always occur and problems that are specific to one run.

To improve the random selection, we added a heuristic: The idea is to prefer short idle times for the aircrafts. I.e. after an arrival, the aircraft takes the next flight as soon as possible. The probability for selection of aircraft a_i with idle time t_i is

$$P(a_i) = \frac{t_i}{\sum_j t_j}$$

If one idle time is twice as big as another, then the chance for selection is only half.

4.3.2 Early Backtracking

Without any further modification, this algorithm will only backtrack when there is no other possibility. The bad thing about trying all possibilities is that in practice it often does not backtrack deep enough, if the critical decision is located too far behind. That is why we may want to backtrack even before we tried all possibilities, so we get more diversity. On the other hand, we might miss a good solution because we backtrack early.

So, the question is what is more important? Our idea was that there must be many possible solutions, and thus it is not so bad if we miss one, but it would be much worse if we spent a lot of time searching for a solution where there is none.

It seemed that the main weakness of this algorithm is the way it chooses the aircrafts, so we searched for better heuristics and implemented a new algorithm, Zoolander.

[1] Finding an aircraft is equivalent to finding a subgraph, because each aircraft has exactly one subgraph where it currently "flies".

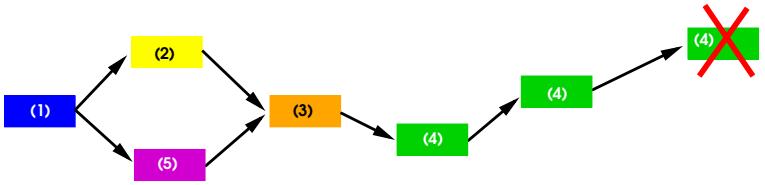

Figure 4.2: Searching the same twice.

4.4 Zoolander

Zoolander is very similar to Shrek. The difference is how the aircraft is chosen. First we should have a closer look at our problem. The main objective is to avoid the time-consuming backtracking. A second goal is to identify as early as possible blind alleys.

4.4.1 Avoiding Backtracking

To cope with the first issue it is essential to analyze the reasons for backtracking. Backtracking is necessary if a subgraph has no longer a path from the start to the end maintenance cluster after some cluster was assigned to another subgraph (and therefore erased in the current subgraph). It might be reasonable to choose the subgraph in such a way that the possibility that one of the others will break after erasing the cluster is as small as possible. This brought us to the idea of considering the minimum cut. The minimum cut endows the number of disjoint paths in a graph respectively subgraph, in the case every edge has the value one. If a subgraph has a minimum cut bigger than one, there will still be a path after erasing an arbitrary cluster. This led us to the heuristics of choosing the subgraph with the smallest minimum cut value.

4.4.2 Earlier Identification of Dead-ends

When having a closer look at the structure of the subgraphs one notices that they are very well connected. Fig. 4.2 gives us an understanding why this leads to difficulties. Let us assume that the blue (1) cluster was the last allocated cluster. Now the yellow (2) cluster will be allocated and all the following up to the red cross, which marks a breaking of a subgraph. So there will be backtracking back to the blue (1) one. In the next step, the purple (5) one will be chosen, and all the succeeding again up to the dead-end. But this was not necessary. After allocating the orange (3) one it should have stopped, because one reached the exact same situation as before, when the orange (3) cluster was allocated the first time. As a conclusion we realize that it is useful to memorize already encountered situations.

Other analysis of the behavior of backtracking produced Fig. 4.3. On the x-axis the clusters are listed in order of their processing. The first cluster on the left is allocated first. On the y-axis the number of attempts to allocate a cluster to a subgraph are shown in logarithmic scale. Because of backtracking, clusters are often allocated more than once. The vertical red lines mark the maintenance clusters.

Before the algorithm reaches the maintenance clusters, it has to backtrack a lot. It seems that clusters directly in front of the maintenance events constitute a bottleneck. Hannibal was created to tackle this problem. Simultaneously, Rainman was developed, following a completely different approach.

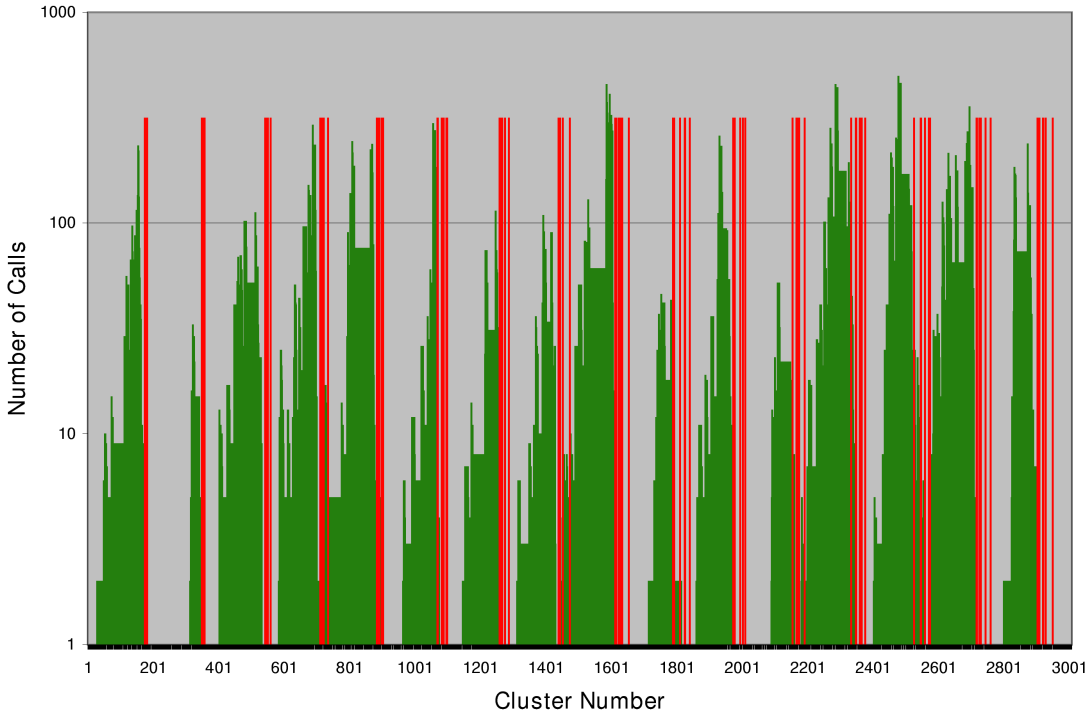

Figure 4.3: Behavior of backtracking

Zoolander

1. Get the first/next cluster, sorted by departure time.
2. Search the valid aircraft for leg ℓ whose subgraph has the fewest node disjoint path (smallest min cut value).
3. Delete this cluster in all the other subgraphs (if it is contained).
4. Go on with step 1.

Backtracking is necessary if a subgraph breaks during step 3.

4.5 Hannibal

Hannibal revives the idea of the first attempt. The clusters are not in a strict order, like they were in Shrek and Zoolander. The basic idea of this algorithm is putting the clusters into the subgraphs, starting at both ends.

Hannibal

1. Mark the start and the end maintenance clusters in all subgraphs.
2. Calculate for all neighbors of all marked clusters the time between arrival and departure of the marked clusters, respectively the other way round.
3. Take the cluster with the smallest time, assign it to the subgraph, and mark it.
4. Delete this cluster in all the other subgraphs (if it is contained).
5. Delete all clusters in the subgraph which are not reachable from the chosen cluster.
6. Go on with step 2.

Backtracking is necessary if a subgraph breaks during step 4 or if a cluster is deleted in all subgraphs after step 5.

4.6 Rainman

Rainman has not evolved from the other algorithms, but follows a completely different approach. The idea behind it is to see as early as possible if an assignment from a leg to an aircraft leads to a problem. To achieve this, we will reserve needed resources when doing an assignment. If a reservation is not possible, because there are already too many reservations, then we cannot do the assignment that caused those reservations. This procedure is also known as constraint propagation [3]. Let us look at the example in Fig. 4.4. These nodes are part of a bigger graph.

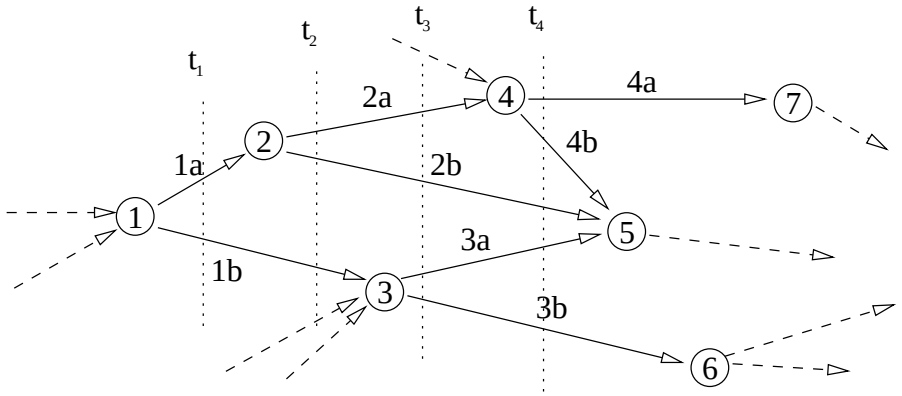

Figure 4.4: A cut-out of a bigger graph

For now, we look only at the reservations of one single aircraft.

In the preceding step, our aircraft arrived at node 1. Think of the aircraft as a flow through the graph — the reservations represent the conditions for every dotted line in the graph. They are as follows: The aircraft must fly

- At t_1: Either edge **1a xor 1b**
- At t_2: If we take 1a, then we need either 2a or 2b, else we need 1b. So we need **(2a xor 2b) xor 1b**

- At t_3: Nodes 2 and 3 cannot be flown both, so **(2a xor 2b) xor (3a xor 3b)**
- At t_4: Now it becomes more complex: **(4a xor 4b) xor 2b xor (3a xor 3b)**
- ...

„xor“ is *exclusive or*, which means that we can choose *either* the first *or* the second, but *not both*. When more than two variables are connected by xor, i.e. A xor B xor C, only one of these can be true. You may wonder why we used parentheses in the expressions above, although they are mathematically unnecessary. We grouped the edges that have the same source, because the algorithm works on these *groups*, rather than on single edges. They are formed by grouping at every time step t_i all edges which have the same source.

In the next step, we choose a succeeding node from node 1, e.g. node 3, and assign it to our aircraft. This means we annul the reservations for all other succeeding nodes of node 1, in this case this is node 2. So we change the reservations:

- At t_1: Reservation now obsolete
- At t_2: Only **1b** is reserved
- At t_3: **3a xor 3b**
- At t_4: **3a xor 3b**

As you can see, when we assign a node, we do not reserve *more*, but we *annul* existing reservations. This annulation needs to be propagated through the graph. In the example above, you can see that node 4 can not be flown anymore, so its outgoing edges[2] are annulled too. Also note that we always annul such *groups* that were defined above, not single edges.

The term “annul node A” is used frequently. This means we annul all groups that contain edges from node A[3], and then propagate the annulation. In the example above, you could also say that we annulled node 2 and then propagated the annulation to node 4 and 7 (but not to node 5, because the aircraft can still reach node 5)

Rainman

1. Initialize reservations
2. Choose the first/next leg l
3. Choose an aircraft α which can fly leg l
   (a) If there are no more aircrafts, backtrack to last leg
4. Do the necessary annulations for aircraft α flying leg l
   (a) If these annulations are not possible, go to (3)
5. Go to (2)

[2] “its outgoing edges” means all groups that contain the outgoing edges, to be correct.
[3] Remember that in a group of edges, all edges have the same source.

4.6.1 Initialization

We have to initialize the reservations such that:

1. An aircraft can not fly maintenance legs of other aircrafts
2. An aircraft must fly all of its own maintenance legs

To do this, we first reserve all groups of edges for all aircrafts. Then we annul for aircraft a_i

- All legs that can not be reached because aircraft a_i must fly one of its maintenance legs
- All maintenance legs that can not be flown by aircraft a_i because they are already assigned to other aircrafts.
- Groups of edges that would allow to bypass a maintenance leg (Fig. 4.5)

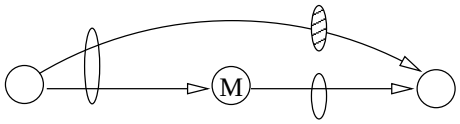

Figure 4.5: The hatched group must be annulled, else it would allow the aircraft to bypass its maintenance leg 'M'.

When all this is done, we are ready to assign the first leg to an aircraft.

4.6.2 Annulling a Node

When a node is to be annulled, we need to propagate the annulation forward and backwards in the graph. While doing this, it always has to be assured that there are not too many cancellations. Too many cancellations means that it is not possible to fly all legs. Now we will discuss how this is to be done exactly. But first, we need a few definitions, to simplify further descriptions:

- The source node of a group is the node which is the source of all edges in the group. Remember that all edges in a group have the same source by definition.
- An edge is called *annulled*, if the smallest group that contains this edge is annulled.
- A node is called *annulled*, if the group that contains all its outgoing edges is annulled. For example, in Fig. 4.6, node A is called annulled if group A1 is annulled. In this case also all other groups that contain edges from the same source must be annulled (in the example: A2, A3,. . .)

Propagate Annulation Forward

The first thing to do is to annul all the groups that contain the outgoing edges. Then, it must be checked for every succeeding node, if all incoming edges are annulled now. If this is true, this succeeding node is annulled recursively.

Propagate Annulation Backwards

Let us use Fig. 4.6 for illustration. Assume we are annulling node C. The annulation is already propagated in the forward direction, so C1 is already annulled. A2 can now be annulled, if A3 is annulled. If we annulled A2, then in the next step, A1 can be annulled if B1 is annulled. If we annulled A1, then we must annul node A recursively. These rules above are those for the first (the upper) incoming edge

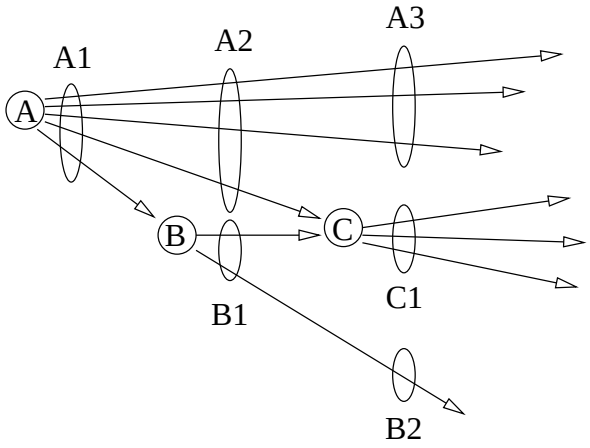

Figure 4.6: Illustration for the conditions for propagating an annulation backwards. Example: A2 must be annulled if A3 and C1 are annulled.

of node C. Of course, this has to be done for all incoming edges of node C.

Checking the Reservations

The problem is kind of a marriage or matching problem. It can be solved as described in [4] by iterative algorithms and has a complexity of $O(n^{1.5}(m/\log n)^{0.5})$. Each aircraft a_i must be assigned to one of a number of groups g_j, while $|a| <= |g|$ (there are at least as much groups as aircrafts). Not all groups need an aircraft, but only those who contain all edges from their source node. These groups are called the *must-groups*. In Fig. 4.6, for example, the must-groups are A1, B1, C1. It is not possible to assign multiple aircrafts to the same group, because this would mean that two airplanes fly the same leg.

If there is no solution for the marriage problem, then the annulation cannot be processed.

Walking Through the Graph

Finally, we use the parts described above to walk through the graph searching a path for every aircraft. The nodes of the graph are assigned one by one to the aircrafts. It is only possible to assign a node to a specific aircraft, if the node has an edge to the last assigned node of this aircraft, and of course the needed reservation must be possible (see above). When a node cannot be assigned to any aircraft, backtracking is needed.

Chapter 5

Optimization

Now that an arbitrary solution has been found, the next task is to find a better, preferably the best solution. The quality of a particular solution is expressed by the costs, which are composed of many things, such as empty seats or violations to the maintenance rules. To lower these costs, the tail assignment will be changed, without breaking the hard constraints. This is far from trivial. The first reason for this is the amount of possible changes: The number of possible changes in a particular solution is typically about 1000 in our case, for a problem with 10000 legs and 25 aircrafts in 5 different versions. Most of these 1000 changes can be combined, so we have at least about $2^{1000} \approx 10^{300}$ possible solutions as a first rough estimate.

The main target is now to find a good tail assignment in feasible time. Before explaining different strategies to achieve this, let us first look at general technical aspects of the optimization process in more detail.

5.1 Terms and Definitions

5.1.1 Neighborhood

There are basically two types of operations we perform to create a new solution. First, legs can be moved from one aircraft to another. Second, version changes can be added or removed between two legs. In our work, we concentrated on the first type of operations, ignoring the possibility of version changes. Different aircrafts can have different versions, but an aircraft cannot change its version.

A situation where two aircrafts are at the same airport at the same time, is called an *active swap*. A swap consists of 4 legs, i.e. 4 nodes in the cluster graph that are connected like in Fig. 5.1. If there are other legs in the solution between these 4 legs in the swap then the swap is called *inactive* (see Fig. 5.2). It is also inactive when the two legs on the right of Fig. 5.1 are not flown by the same aircrafts as the two legs on the left. In other words, either the crossed or the parallel edges in Fig. 5.1 must be part of the current solution.

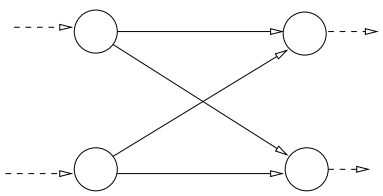

Figure 5.1: An active swap in the cluster graph.

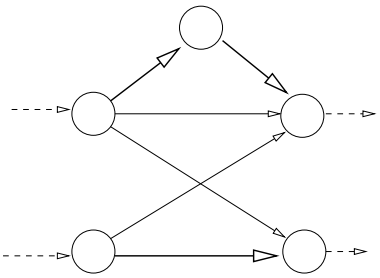

Figure 5.2: An inactive swap in the cluster graph. The bold arrows represent the current solution. As soon as the top node is flown by another aircraft, the swap consisting of the lower four nodes becomes active.

In such a swap situation, two aircrafts can exchange all succeeding legs. But this is not what we want, because this would mean that the aircrafts arrive at other maintenance events, however this is not allowed (maintenance events belong to a specific aircraft, this cannot be changed). To overcome this, we need always *two* swaps in the same subgraphs, such that the second swap leads the aircrafts back to their predefined maintenance events. This way it is guaranteed that the aircrafts still arrive at their maintenance events. Applying two swaps in this way is called an *exchange*. These exchanges lead to the neighbors as we defined them above.

A closer look at these neighbors shows the following important properties:

1. It is not possible to repeatedly "move in a certain direction". Doing the same operation (exchange) twice means nothing else than undoing it.

2. The number of neighbors is not fixed. This is because when moving legs from one aircraft to another, new swaps can become active, and active swaps become inactive.

To understand point one, one must understand how the solution space, defined by our neighborhood, looks like. In a typical, 2-dimensional continuous solution space, for example, one may try to move to the same direction, as long as it improves the solution. This is not possible in our solution space, because an exchange can either be done, or not. So we have a high-dimensional binary solution space, which explains the first property above.

5.1.2 Cost Calculation

Costs can arise from the following situations:

- Spilled passengers
- Empty seats
- Version changes
- Violation of the maintenance check rules

It is obvious that different weights must be assigned to these situations, because spilled passengers are for example much worse than empty seats, or an empty first class seat causes more costs than an economy class seat. We choose the weights in our model according to Tab. 5.1.

The total costs are the weighted sum of the single costs

$$C_{tot} = \sum_i a_i \omega_i$$

Description	Symbol	Weight ω_i
Spilled passengers		
First	a_1	12
Business	a_2	8
Economy	a_3	5
Empty seats		
First	a_4	3
Business	a_5	2
Economy	a_6	1
Other		
Version changes	a_7	50
Exceeding cycles	a_8	75
Exceeding flight hours	a_9	5

Table 5.1: Weights for the calculation of the total costs

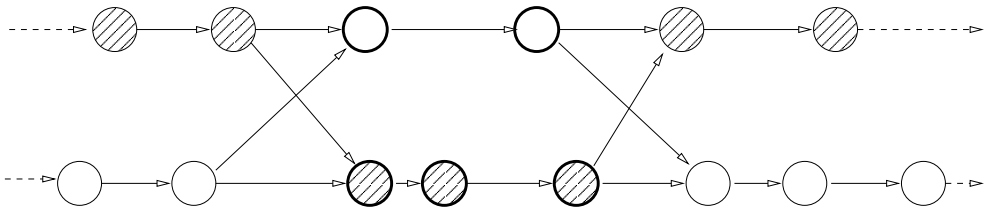

Figure 5.3: Leg exchange. Before the exchange, the upper legs belong to aircraft 1, the lower ones to aircraft 2. After the exchange, the hatched ones belong to aircraft 1, the others to aircraft 2. The bold clusters are those who are exchanged.

where ω_i are the weights from Tab. 5.1, and a_i denotes the amount, i.e. spilled passengers, empty seats, version changes or the number of cycles/hours by which the check rules are violated.

5.1.3 Exchanging Legs

In Fig. 5.3, a situation with two swaps, as described in Sec. 5.1.1, is illustrated. To actually perform the exchange, the legs whose assignment changes must be found (those are bold in Fig. 5.3). Then these legs are moved to the other aircraft.

When the legs are assigned to their new aircraft, the active swaps must be adopted to this new situation. Swaps that include two legs that are no longer assigned to the same aircraft will become inactive and vice versa.

5.1.4 The Problem with Planes

The number of aircrafts usually is much greater than the number of aircraft versions. Therefore, a lot of equal airplanes exist, which produce the same costs when performing the same clusters. Hence, during the optimization process many of the neighbors of a current solution have equal costs. Metaphorically speaking there is a “plane” in the solution space. Here, a plane means solutions with equal costs that are neighbors. In other words, two solutions are in the same plane if it is possible to get from one to the other passing only solutions with equal costs.

These planes need special attention from optimization algorithms like local search. If an algorithm meets a solution whose neighbors have all equal or higher costs, it is not certain which neighbor leads to the local minimum. Therefore it is necessary to check all neighbors of all solutions of this plane. The navigation on this plane is not simple. It is essential to avoid cycles, that means not passing the same solution twice, and it should be possible to assert that the whole plane is searched, in the case that the plane is **not** a local minimum.

5.2 Local Search (LS)

Local search is the simplest but nevertheless often successfully applied method for searching the minimum. All neighbors are considered and the one with the lowest costs is chosen. This is repeated until no neighbor has lower costs then the current solution (see e.g. [5]). This is of course easy to implement, but a disadvantage is that each time the costs of all neighbors have to be calculated.

The planes described in Sec. 5.1.4 need some attention. There are two possibilities to handle this circumstance. If LS meets one solution of such a plane, just stop and take these costs as an approximation of the costs of the real minimum. The second possibility is to search the whole plane for a neighbor with lower costs. *Depth-first search* should be able to solve this problem. Take care of cycles!

5.3 Tabu Search

A way to find an exit from the numerous local minima is tabu search (see [5]). The basic idea is to forbid exchanges that were already performed recently. These exchanges are *tabu* for a certain time, hence the name of the algorithm. All current tabu exchanges are in the so called *tabu list*.

An important parameter is the size of the tabu list or, in other words, for how long a move is tabu. In our model, this should not be constant, because the number of neighbors is also not constant. Instead, the size of the tabu list is defined relative to the number of neighbors. Another parameter is needed to stop the optimization when the best known solution did not improve over the last t_{stop} iterations of tabu search.

Tabu Search

1. Calculate cost of all neighbors
2. Choose the best neighbor which is not tabu (Either the move is not in the tabu list, or it meets some “aspiration criteria”)
3. Move to the chosen neighbor
4. Add this move to the tabu list
5. If tabu list is longer than some t_{len}, remove the oldest element.
6. Iterate until the best known solution did not improve over some predefined number of steps t_{stop}.

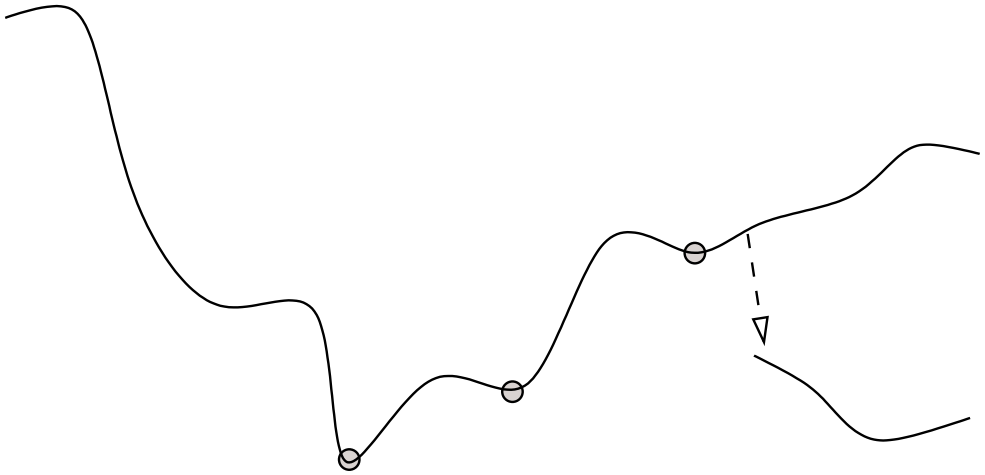

Figure 5.4: Cost development during the optimization process (from left to right). The last local minima are marked. Overriding the tabu restriction is symbolized as a dashed arrow.

5.3.1 Aspiration Criteria

Aspiration criteria are introduced in tabu search to determine when tabu restrictions can be overridden, thus removing a tabu classification otherwise applied to a move. In the first attempt, we used a simple criterion, saying a tabu restriction can be overridden if the cost after the exchange are lower than the best known cost so far.

This aspiration criterion was used frequently only at the beginning of the optimization process, but not later on, when things really start to get interesting. A more sophisticated idea is to override the restriction already when the solution improves compared with the last local minimum. Fig. 5.4 illustrates this. A list of the last minima is maintained, containing the three marked minima in Fig. 5.4. Notice that the minimum on the left is not more in our list, thus not marked, because we remove all minima from the list that are greater than our current solution. This way, we end up with a list of increasing minima, the first being the lowest, best known solution yet.

Local Minima

Local minima are detected during the normal optimization process, without explicitly looking for them. The rule says there is a local minimum when

1. The cost rise in the next operation
2. No tabu operation could lower the cost

This is no guarantee for a local minimum, it is also possible that a plane in the solution space is found and misinterpreted as a minimum (see Sec. 5.1.4).

5.4 Simulated Annealing (SA)

The basic structure of *simulated annealing* is very simple. [5] It works similar to LS. The difference is, that SA accepts not only solutions with lower costs. Therefore it is possible to leave a local minimum and search in a wider area. The probability that a neighbor with higher costs is accepted, is decreasing over the time.

This is done by a temperature T. A neighbor is accepted if the costs $C(x')$ — where x' is the neighbor of x, the current solution — is lower then $C(x)$. Otherwise it is accepted with the likelihood p:

$$p = \exp \frac{C(x) - C(x')}{T} \tag{5.1}$$

The challenging part is to find a well performing function to update the temperature and to find good bounds for its range.

The algorithm is described in the following box:

Simulated Annealing

1. Let x be the current solution, take an arbitrary neighbor x' (selected uniformly at random).
2. If $C(x) > C(x')$ OR $p > rand[0;1[$ then $x \leftarrow x'$.
3. Update temperature $T \leftarrow next(T)$
4. Iterate until the end temperature is reached.

5.5 Combination of SA and LS

Simulated annealing finds the global minimum if the cooling is done by evanescent steps. But that needs infinite time. To finish the calculations in feasible time the cooling steps need to be of reasonable size.

5.5.1 Local Search with Different Starting Points (LSwDSP)

Let us have a closer look at the solution space and how SA traverses it. Experiments show that there are a huge number of local minima. Therefore the quality of LS depends on the starting solution. Using a bunch of different starting positions is a problem. As we have seen in Chap. 4, it takes a lot of time to find one single starting solution. So we started to think about a way to get different starting points. Now, what is SA doing? In the beginning it travels around in the solution space almost without any restrictions. That means it accepts most neighbors. If the end temperature is not too high it ends in a local minimum. Say $x_1, x_2, \ldots, x_n$ are the accepted solutions of SA during the cooling process. So, what do we have? In the beginning of its process, SA finds many different solutions, not really caring about their costs. Now you can find a local minimum for each of them, using LS. Say $\ell_1, \ell_2, \ldots, \ell_n$ are the corresponding local minima. Finally, we found a way to get different starting points.

It is probable that a good local minimum is missed in the beginning of the SA process and that there is no return because of fast cooling. Instead of slowing the cooling, one can calculate for each x_i the corresponding ℓ_i and save the lowest.

5.5.2 Chained Local Optimization (CLO)

Oliver C. Martin and Steve W. Otto had a similar idea as we had. Their attempt is described in [6]. CLO is also a combination of SA and LS. The basic idea is the following. Instead of comparing $C(x)$ and $C(x')$ to decide for or against the neighbor

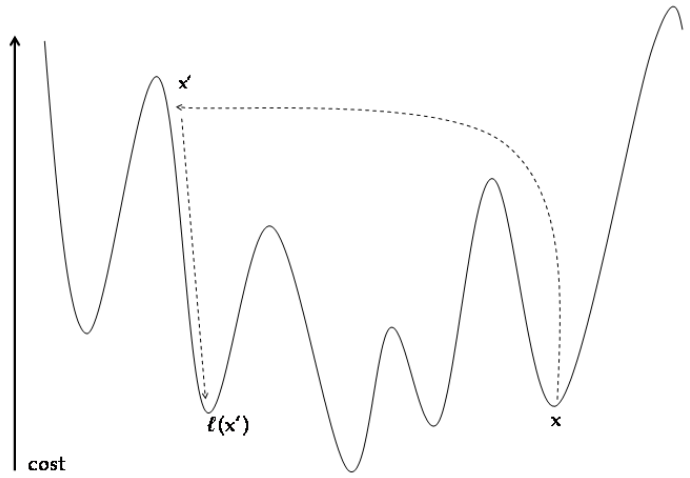

Figure 5.5: Illustration of a CLO-step from x over x' to $\ell(x')$.

they compare the corresponding local minima $C(\ell(x))$ and $C(\ell(x'))$. If the new value is accepted not x' is the new starting point but $\ell(x')$. A problem is that in a lot of cases all the neighbors of x have the same corresponding local minimum $\ell(x)$. The result is that it makes no headway. Therefore it is necessary that the new candidate is not a direct neighbor but a "n-step neighbor". That means to take the neighbor of the neighbor of the ... Fig. 5.5 illustrates one CLO-step. The algorithm is described in the following box:

Chained Local Optimization

1. Take an arbitrary n-step neighbor x'.
2. Use Local Search to find the corresponding minimum $\ell(x')$.
3. If $C(x) > C(\ell(x'))$ OR $p < rand[0;1[$ then $x \leftarrow \ell(x')$. (p is defined according to (5.1).)
4. Update temperature $T \leftarrow next(T)$
5. Iterate until the end temperature is reached.

Chapter 6

Results and Conclusions

6.1 Initial Solution

6.1.1 Shrek

The use of early backtracking, as explained in Sec. 4.3.2, improved the Shrek algorithm, but one more parameter need to be adjusted: The probability p_{back} for backtracking after an unsuccessful assignment. If p_{back} is (too) high, then Shrek happens to backtrack back to the first leg. In this case, no solution is found, Shrek *fails*. Trying many values of p_{back} to find an optimum would not be very useful, because the optimal value is different for every data set. Instead, when Shrek fails to find a solution, p_{back} is decreased by 0.02 and the Shrek algorithm is started again. This way, p_{back} will be decreased until Shrek does not fail anymore.

When choosing the aircrafts using the described heuristic from Sec. 4.3.1, the runtime of the algorithm is very irregular. If one has the possibility to run multiple instances in parallel, several Shrek instances can be started, and when a solution is found by anyone of them, all can be terminated. There is still no guarantee for a fast solution this way, but it is more likely to find a solution soon.

6.1.2 Zoolander

Zoolander has the ability to find a solution for the most generated data sets. The choice of the min cut as the criterion how the subgraph (see Sec. 4.4.1) is chosen had reliability entail. The method for the early detection of dead-ends explained in Sec. 4.4.2 brought a speedup of 40%. Bigger data sets, i.e. many clusters (more than 5000) and especially many aircrafts (more than 10), need too much time (more than a day).

6.1.3 Hannibal

Hannibal is not completely implemented yet. At the moment it is able to find an initial solution that is correct — every string contains clusters that can be flown in order — but not all clusters are allocated. But the speed is impressive. Hannibal solves some data sets in a few minutes where Zoolander needs more than an hour. There is a good chance that Hannibal will still be faster then Zoolander even if the must of allocating every cluster is respected.

6.1.4 Rainman

The difficulties of this algorithm boil down to the so-called marriage or matching problem: Given a aircrafts, g groups of legs and $a <= g$, assign every aircraft to a different group.

Rainman is implemented except for this matching problem. We tried a very basic implementation which tests only that an aircraft can be assigned to every first group succeeding each node. This means it tests only if every leg can still be reached by an aircraft. This way it was possible to solve small problems, but of course such a basic implementation does not show what the algorithm could do otherwise. We stopped the implementation of Rainman, or more precisely the matching algorithm, because it would have consumed a lot of our limited time for this thesis. Instead we investigated the optimization algorithms.

Nevertheless, some things can be said about important things in this algorithm: The initialization and the matching problem.

Initialization

Unlike the other algorithms, Rainman needs a lot of initialization before doing the first assignment. This needs time, but more important is, it needs memory. The memory consumption depends strongly on the length of the edges in our graph, or, in other words, how long an aircraft can possibly stay at an airport.

Matching Problem

For every annulation, the matching problem needs to be solved several times, depending on how far the annulation needs to be propagated. It must be solved so many times that the performance of Rainman mainly depends on the performance of the matching algorithm!

6.1.5 Comparison

It is difficult to compare Shrek and Zoolander, because Shrek's runtime is very irregular and for a given problem size, both algorithms behave differently for different problem instances (with the same parameters), as can be seen in Tab. 6.1. All data sets use the same parameter values, but some seem to be "simpler" than others. Both algorithms seem to have difficulties with the same problems. Data set 4 for example, was solved in 5 of 6 runs in less than 2 minutes, while data set 5 was never solved in less than 6 minutes.

While Shrek is generally faster than Zoolander for the problems in Tab. 6.1, Zoolander is more reliable for more complex problems. Often Shrek does not find a solution in feasible time for these problems. An example is given in Tab. 6.2 where Shrek only found a solution in one of five runs.

There are also problem instances with the same parameters as in Tab. 6.2, that could not be solved, neither by Shrek nor by Zoolander (3.5h runtime). Our assumption is that these problems happen to have only few solutions, so finding one of them is very difficult.

6.2 Optimization

There are basically two points we want to examine. The first point is the quality of the solution the optimizer produces. I.e. answering the question: How high are

Data set	4-10-7-3-1	4-10-7-3-2	4-10-7-3-3	4-10-7-3-4	4-10-7-3-5
Zoolander	89.2s	1816.0s	1251.3s	112.0s	1466.0s
Shrek run 1	54.4s	147.1s	134.9s	60.1s	1661.8s
Shrek run 2	521.1s	2019.9s	22784.4s	89.8s	1476.6s
Shrek run 3	688.9s	388.1s	458.7s	160.5s	409.3s
Shrek run 4	91.9s	1736.0s	490.7s	no solution	2521.2s
Shrek run 5	335.7s	146.5s	285.7s	37.6s	1844.5s

Table 6.1: Runtimes for finding an initial solution on a AMD Athlon XP 2800+, using Shrek or Zoolander. Shrek was limited to 50 million iterations. The data sets were generated with 4'000 legs, 10 airports, and 7 aircrafts in 3 versions.

Data set	4-35-7-3
Zoolander	156.7s
Shrek run 1	no solution
Shrek run 2	no solution
Shrek run 3	4243.1s
Shrek run 4	no solution
Shrek run 5	no solution

Table 6.2: Runtimes for finding an initial solution on a AMD Athlon XP 2800+, using Shrek or Zoolander. Shrek was limited to 50 million iterations(about 3.5h runtime). The data sets were generated with 4'000 aircrafts, **35 airports**, and 7 aircrafts in 3 versions.

the savings of costs? The second point is the calculation time. Therefore, how long does it take to get these costs?

6.2.1 Test Methods

We generated several data sets to compare the different optimization methods. Tab. 6.5 shows the four data sets with their parameters. The columns 2 to 5 show the number of *clusters* , *airports*, *aircrafts*, and *versions*. The basic data set is **Set A**. **Set B** has half as many clusters, **Set C** has half as many airports and finally **Set D** has half as many aircrafts. In **Set D** is the number of versions suited to the number of aircrafts, which is five times as much. The reason why we used these data sets is, because we do not really know what data the real world makes available. So we think it is possible to gather from the evaluation of these four data sets to the most others. We used the data from Lufthansa as an indication.

Because of earlier tests we know that different data sets with the same parameter values can produce various behaviors during the optimization. These variations are relevant especially for small data sets with less than 3000 clusters. Therefore we created of each data set five instances. Since the calculation time of some algorithms exceeds twenty-four hours we used only two of each data set.

The next sections describe how the different algorithms work on these data sets and what results they produce. The last section compares all of them.

Set Name	Clusters	Airports	Aircrafts	Versions
Set A	10000	40	30	6
Set B	5000	40	30	6
Set C	10000	20	30	6
Set D	10000	40	15	3

Table 6.3: Data sets

6.2.2 Local Search

Local search is not a usable algorithm to find a good solution for our problem. That is because of the topology of the solution space. The number of local minima is so huge that it is pure matter of luck how good the solution is. It depends only on the starting solution.

We used two local search methods. The first one stops as soon as all neighbors of the current solution have higher or equal costs. The second one stops when all neighbors of the current solution have higher costs and all neighbors with equal costs are visited.

Results

The great handicap of LS is that it has to calculate the costs of all neighbors. The number of the neighbors depends on the data set. Of course it is proportional to the number of clusters. Therefore **Set B** has half as much as **Set A**. But more important is the proportion between the number of aircrafts and the number of airports. If the number of airports is much bigger then the number of aircrafts (**Set D**) then the possibility that two aircrafts are at the same time at the same airport is little. Therefore the number of neighbors is small. **Set D** has only a third as much neighbors as **Set A**; and **Set C** has double as much as **Set A**.

Using LS for **Set C** takes much more time then SA with fast cooling (around 50 times), and gets similar costs. The second method of LS gets only unessentially lower costs then the first one but needs around a third more time.

6.2.3 Tabu Search

The main difficulties when using tabu search are to find the optimal values for the length of the tabu list, t_{len}, and the stop criterion t_{stop}, which is the number of iterations in which the best known solution did not improve.

t_{stop} is not very difficult to adjust. When it is too low, optimization will stop very early. This was the case for values below 1000. When t_{stop} is set to infinity, optimization will never stop automatically, but must be stopped manually. The tabu list size t_{len} is more difficult to set. See Fig. 6.2 for an illustration of the behavior of tabu search for different values of t_{len}. When it is set too low, the algorithm will get stuck in local minima. You can see in Fig. 6.2 that the cost development is rather flat in this case. Chances are bad that the algorithm gets out of a minimum. But when t_{len} is set too high, the solution will not improve because the needed operations are tabu. An indication for t_{len} being too high are rare big steps in the cost development. Many runs are needed to find the optimum for t_{len}, which turned out to be somewhere around 0.23, as can be seen in Fig. 6.1.

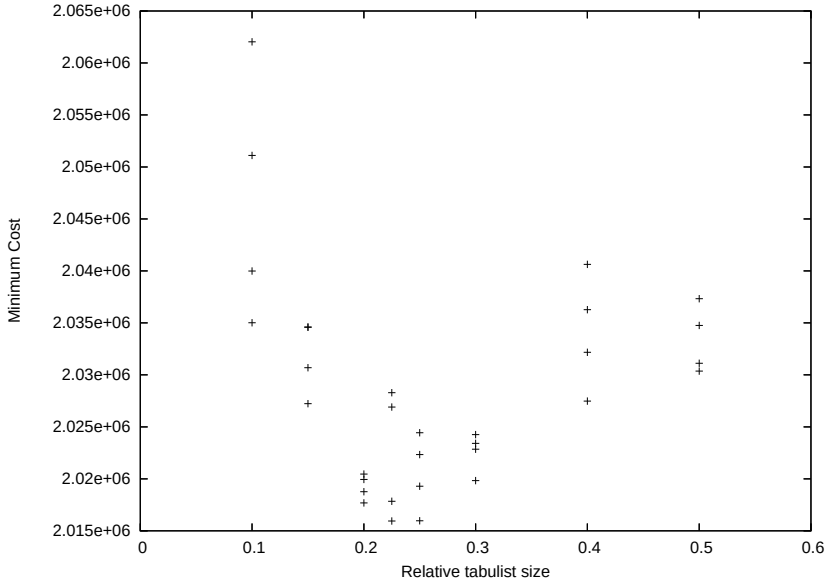

Figure 6.1: Costs of the best solutions found by tabu search. For all runs t_{stop} is 5000.

Tabu search turned out to successfully overcome local minima, but a considerable disadvantage is the need to evaluate the cost of each neighbor at every iteration. This evaluation time is proportional to the number of possible exchanges. The more exchanges are possible, the worse this problem gets.

In Tab. 6.4 are the results of tabu search optimization for data sets according to Tab. 6.5. The first instance of each set is the same that was used for the SA results. Comparing these, one can see that they are always worse than SA with with temperature factor $f =$ `0.999'99`.

6.2.4 Simulated Annealing

Simulated annealing turned out to be a very usable algorithm to solve this kind of problem. The difficult thing is to find a good temperature function. Good means that the result should be as close as possible to the global minimum and the running time has to be feasible. We used the update function (6.1) for the temperature T with the factor f.

$$T = T \cdot f \tag{6.1}$$

T ranges between a start temperature s and an end temperature e. Using (6.1) the number of steps n, the algorithm proceeds, is

$$n = \frac{\ln(e) - \ln(s)}{\ln(f)} \tag{6.2}$$

The choice of the range for T is not sensitive as we will see. You can set the lower limit using the following idea. If the possibility, that a solution, whose costs are 1 higher than the actual costs, is chosen, is around 1‰, it is not possible that it will ever leave the local minimum. Using equation (5.1) you get 1.4 for $T = e$.

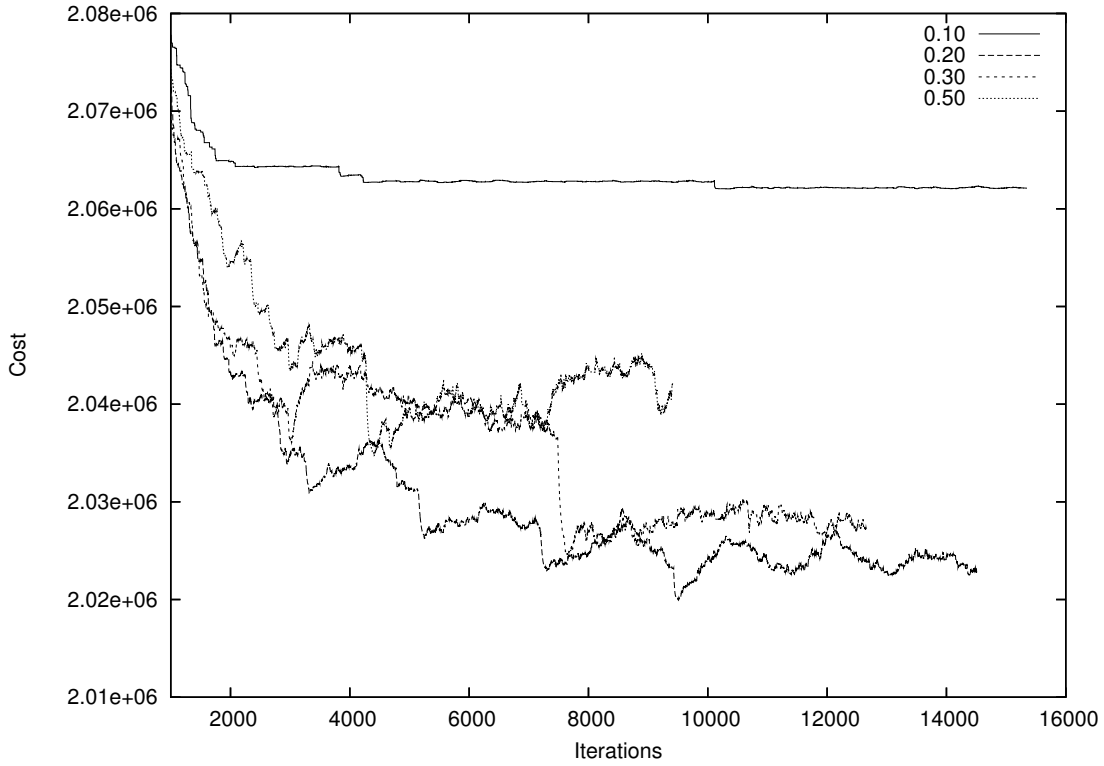

Figure 6.2: The optimization process for different relative tabu list sizes.

To be sure to get the local minimum use local search with the final point from simulated annealing as the starting point. The idea for the upper limit: Say m is the maximal difference of the costs between two neighbors. In the beginning a neighbor with costs that are m higher should be accepted with a probability of around 99%. Therefore s can be calculated with equation (5.1). Choosing the limits as described above, we can be sure that a bigger range cannot produce a better solution.

The temperature factor f gives us enough elbowroom to influence the calculating time and the quality of the solution. If f is close to 1, the quality of the solution will be good but the number of steps will be high. If f is much smaller then it is the other way round. We used four different values for f: `0.999`, `0.999'9`, `0.999'99` and `0.999'999` to optimize the data sets.

Results

The advantage of SA is that the calculation time does not depend on the data set, only on its parameters. Fig. 6.3 shows the optimization steps of simulated annealing optimizing **Set A**. On the x-axis is the temperature displayed starting at 100'000 and ending at 0.2. The scale is logarithmic because the temperature decreases exponentially. The costs are assigned on the y-axis. Displayed are twenty series, for each temperature factor f (`0.999` ... `0.999'999`) five series. One serial shows the propagation of the decreasing of the costs, during the cooling. For each temperature step simulated annealing calculates costs. If these costs are lower than all calculated costs before they are displayed in the serial. Keep in mind that the number of temperature steps in each decade is for each f different. For f = `0.999'9` the number of steps is around 10 times higher then for f = `0.999`. The same for the others.

Instance	Set A		Set B	
	Min. Cost	at iteration	Min. Cost	at iteration
1	2'040'223	7'075	1'047'876	3'354
2	1'785'776	11'516	1'019'233	3'503
3	1'934'624	4'139	915'168	6'282
4	1'939'365	8'751	979'559	12'773
5	2'018'049	3'640	1'058'933	7'061

Instance	Set C		Set D	
	Min. Cost	at iteration	Min. Cost	at iteration
1	1'991'736	9'039	1'631'988	2'167
2	1'864'834	10'293	1'644'437	12'103
3	1'857'965	9'442	1'648'584	2'926
4	1'929'453	8'184	1'553'597	2'896
5	1'892'335	9'616	1'808'409	1'314

Table 6.4: Tabu search results for 5 instances of each set. The first instance is the one that was also used for the simulated annealing tests. Parameters: t_{len}=0.23, t_{stop}=5000

The shape is for all more or less the same. There are two sharp bends in each graph, one around 2'000 the other around 200. The series `0.999` are a bit different, the second bend is not sharp.

The graphs for **Set B** and **Set C** look similar. **Set D** represented by Fig. 6.4 is different. (For each f are only 3 series plotted.) The series `0.999` is good cognizable, but the other series are overlapping. This behavior is the same in all instances of **Set D**. Figs. 6.5 and 6.6 show the minima which simulated annealing found for the four different data sets. Here we sea the statement written before clearer. Only in **Set D** the best result produced a serial with f = `0.999'99`. Therefore it is obvious that the choice of the factor f depends on the data set. If the number of airports is much higher than the number of aircrafts — so the number of solutions is small — the cooling can be faster than when it is the other way round.

Tab. 6.5 shows the calculation time against temperature factor f. The calculations run on a `Sun Fire V60x with 3060Mhz`. With a calculation time less then seven minutes pretty good results can be produced. Investing an hour it is getting close to the best results we have calculated. To top them you need much more time.

Temperature factor f	Calculation time `[h:min:sec]`	Iterations
`0.999`	`0:01:34`	13116
`0.999'9`	`0:06:46`	131217
`0.999'99`	`0:57:27`	1312230
`0.999'999`	`9:02:01`	13122357

Table 6.5: calculation time against temperature factor f.

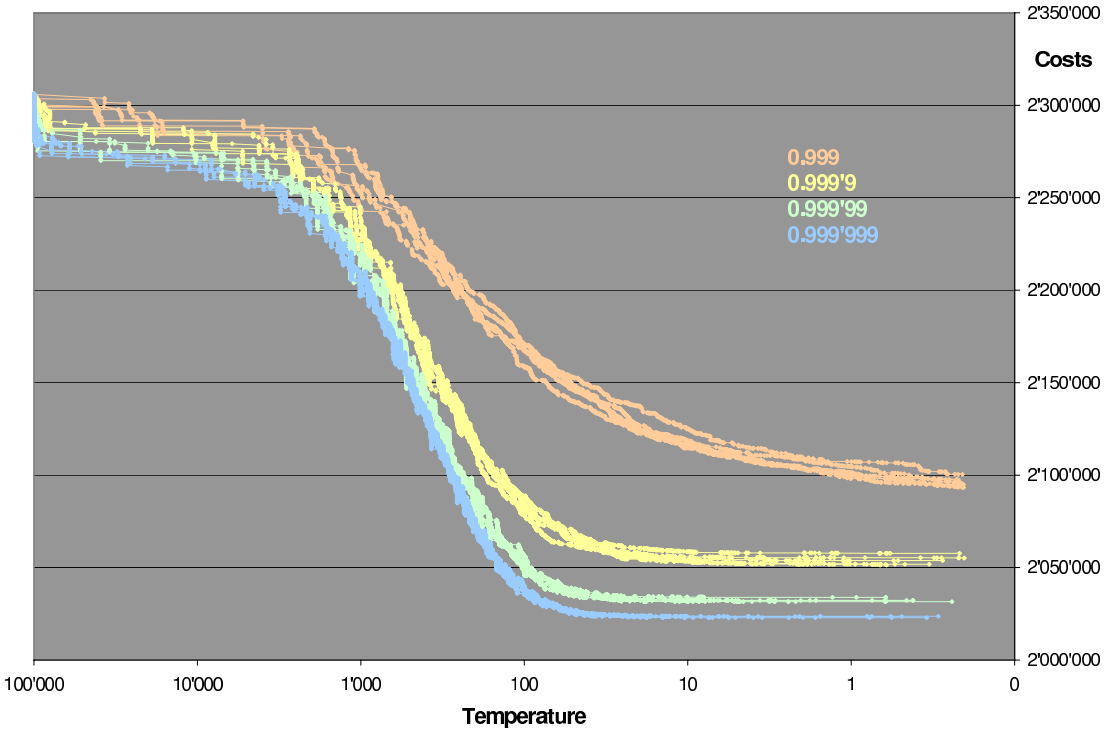

Figure 6.3: Set A optimized by simulated annealing.

6.2.5 Chained Local Optimization and Local Search with Different Starting Points

CLO has the advantage that the cooling can be much faster than by SA. Also the starting temperature can be lower because all the solutions, because of the used LS, have costs close to the reachable minimum. The number of SA-steps n has to be decided. We tested values between 1 and 10 steps. It turned out that a value in the middle delivers usable results, so we took 6 for our calculations. The temperature range was `[100 ... 0.2]`. f has been set to `0.999`. The results have always been worse than the ones of SA with f = `0.999'99`.

If you are interested not only in better results than simulated annealing produces but also in similar calculation time LSwDSP cannot use slower cooling rates than `0.999'9`. With this adjustment it is almost as fast as SA with f = `0.999'99`. As written in Sec. 6.2.2 the calculation time depends on the data set. The results have been worse than SA with aforesaid factor f for all data sets.

6.2.6 Conclusions

A common weakness of the LS and Tabu algorithms is that they are very slow because they evaluate the cost of all neighbors at every iteration. Tabu search reaches only about 1 iteration per second, for a neighborhood of about 1000 on a SunFire V60x (Intel Xeon 3.06 GHz CPU). Simulated annealing reaches about 400 iterations per second, independently of the data set. We will give suggestions how the speed can be increased in Sec. 7.2.2.

Due to the results written above it is obvious that local search attempts are of little value. This is because of the topography of the solution space. Like the name says it only acts in the local environment and therefore it stumbles on the huge number of local minima.

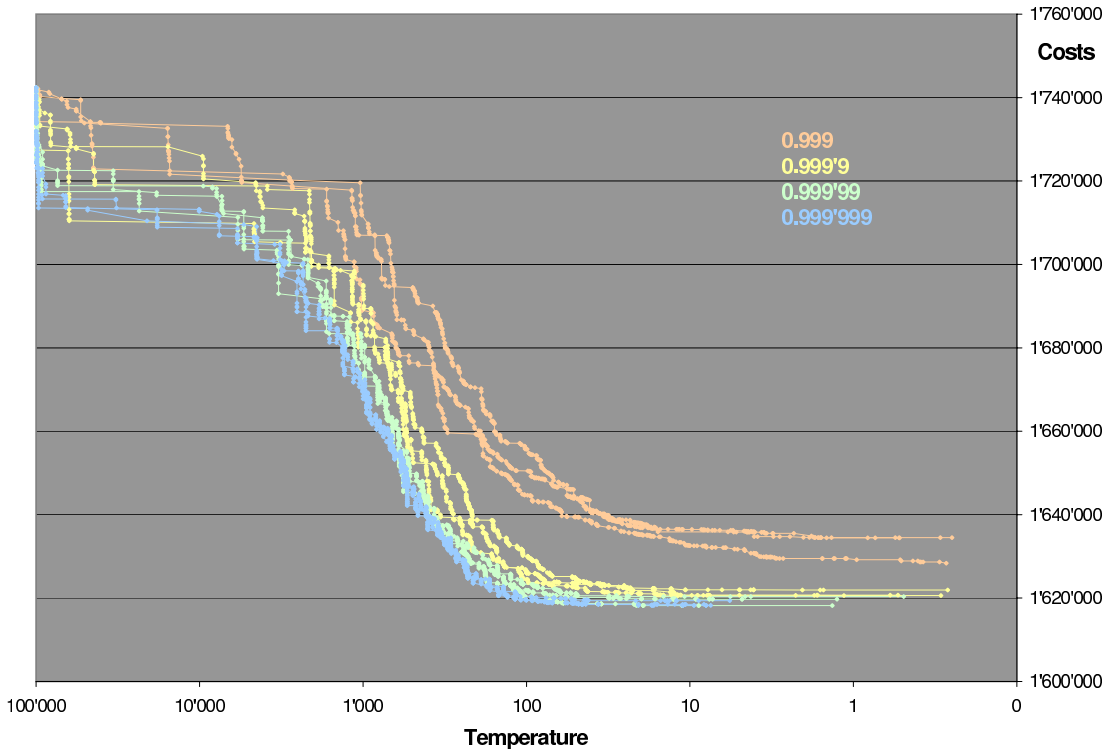

Figure 6.4: Set D optimized by simulated annealing.

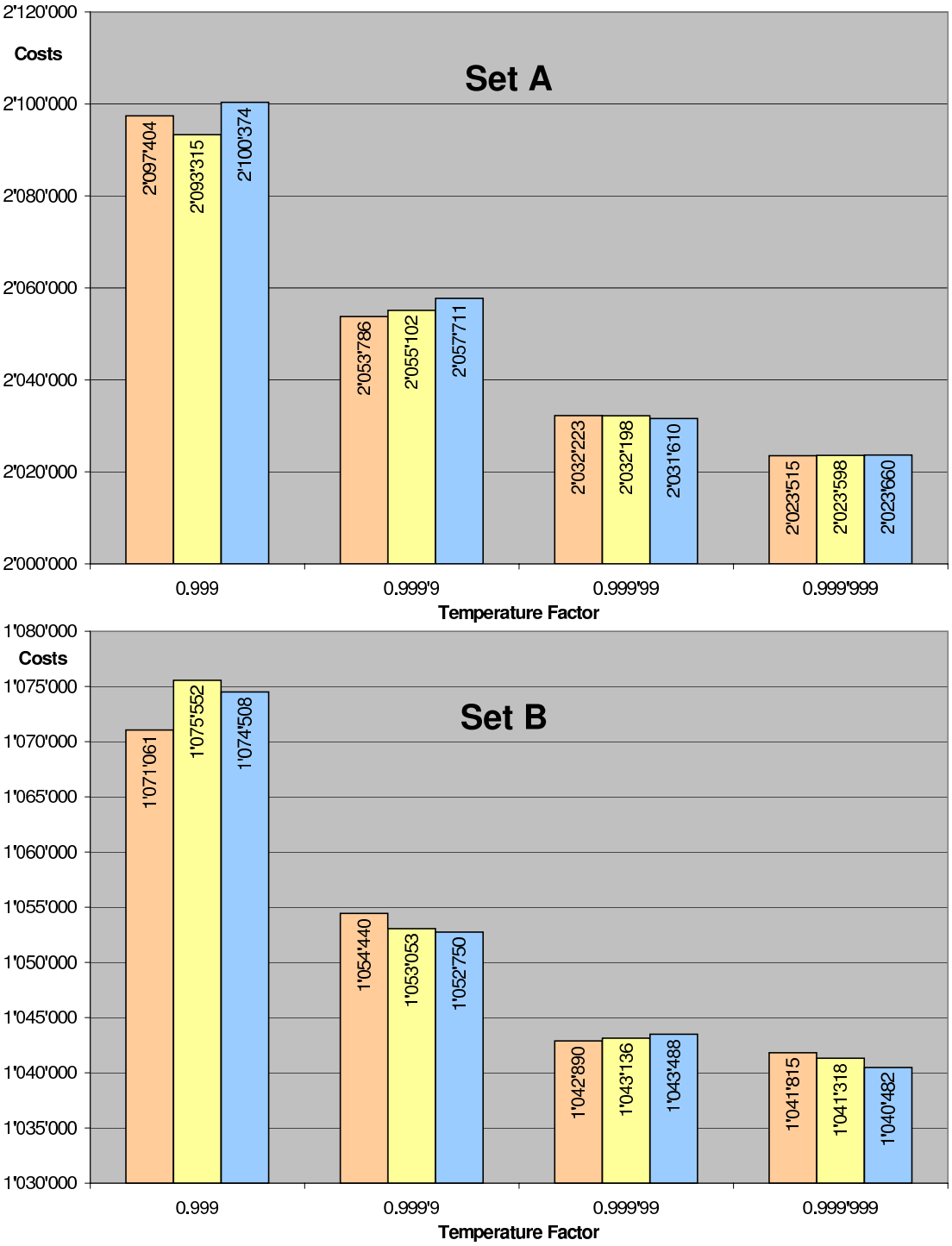

Figure 6.5: The minimum costs of Set A and Set B found by simulated annealing.

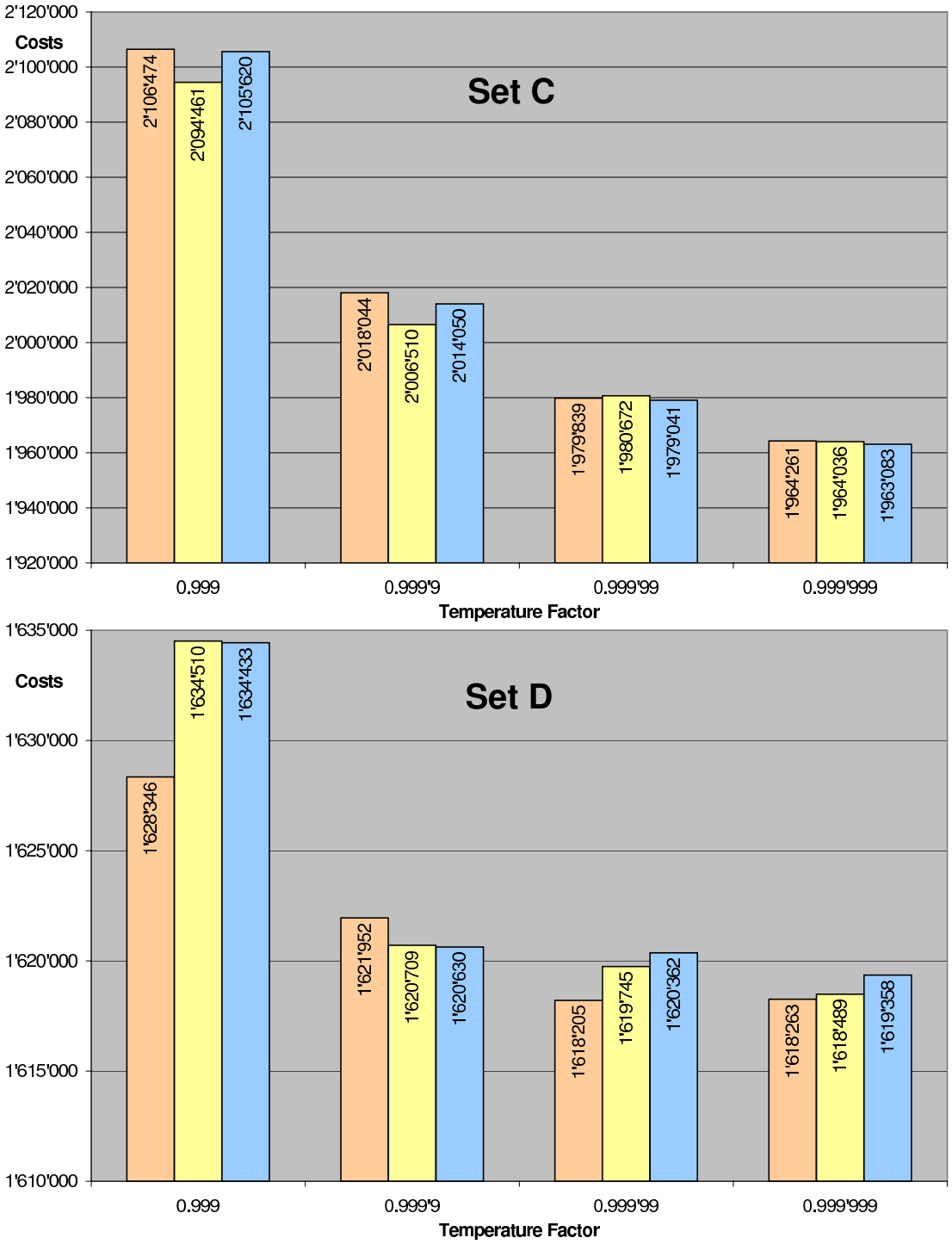

Figure 6.6: The minimum costs of Set C and Set D found by simulated annealing.

Chapter 7

Outlook

We only had six months to finish this thesis, but with all the ideas we have now, we could work on it for another six months! This chapter will show where it is worth to spend more time.

7.1 Initial Solution

The algorithms **Hannibal** and **Rainman** are not completely implemented and tested. Hannibal has the ability to solve some data sets much faster then Zoolander or Shrek, but without allocating all clusters. So it is worth to complete the implementation because there is a good chance that it will still be faster than the others.

Rainman is surely also worth a complete implementation, because it is based on a new idea which holds different pieces of information about the graph than the other algorithms. There is also much room for improvement, as for example by saving the groups in another way. One could save a lot of memory and computation time by implementing subgroups, which are part of another group, just like subsets of sets. Looking at Fig. 4.6 one can see that A2, A3 etc. are all subgroups of A1. In our implementation, A1, A2, A3 etc. are all handled separately.

Another idea for an initial solution finder is an algorithm that takes some soft constraints, like the number of cycles between two maintenance events, into account, so the optimizer can start with a better solution. But the fact that a quite good solution can easily be found by the optimizer shows that it would only add unnecessary complexity to the solution finder. The better way to combine solution finding and optimization is the method with the overflow aircraft described in Sec. 3.3. We have not tried that yet, but Lufthansa follows this approach in their NetLine/Ops optimizer [7]. If there is no way to develop a reliable initial solution finder for real world data sets, this method is absolutely worth having a look at.

7.2 Optimization

7.2.1 Algorithms

Simulated Annealing and Local Search

As we have seen, algorithms which are based on local search are weak compared to simulated annealing. The latter has still some potential. We did not spend much time in the evaluation of temperature functions. It might be difficult to find

a solution with lower costs but it should be possible to find one that is as good as ours but in less time.

Genetic Algorithms

There are other types of optimization algorithms that are of value to have a look at, in particular genetic algorithms. Two solutions can be analyzed string by string and produce a new solution merging these two together and use this solution as a new start solution. It is necessary to take care of conflicts between strings, i.e. every cluster must only be in one string.

Another idea is to start several instances of simulated annealing at a time and stop after predefined time steps. After each of these steps only the ones with the best solutions will go further (again with several instances). This idea seems interesting because it is possible to execute a huge part of the whole solution space. And it is easy to let it run on a multiprocessor system.

It is also possible to combine these two ideas.

7.2.2 Implementation of Cost Calculation

Evaluating the costs of all neighbors of a solution is a very time-consuming process, because the neighborhood can be big. But when looking at the cost of these exchanges, one can see that most of them do not change during an exchange. This is because an exchange will have no influence on another exchange which performs an operation on different subgraphs. Even exchanges which operate on the same subgraphs have no influence to each other, if the maintenance check rules are not exceeded and no version changes are involved. By maintaining more information in the subgraphs, for example the number of flight hours left until the soft limit of the maintenance rules, one could save most of the computation time. The time for evaluating all neighbors would no longer be proportional to the number of neighbors, but only proportional to the number of aircrafts. The result would be a significant speedup for the local search based algorithms. Simulated annealing could not benefit that much, because it does not need the cost of all neighbors.

Appendix A

Data Generator

Lufthansa Systems gave us three data sets with real world data but they have not been usable. Overlapping maintenance clusters and not assigned maintenance clusters made them to garbage. Therefore we used programmed a data generator. The data have exactly the same structure as the ones from Lufthansa.

The input values are:

- Number of airports AP
- Number of clusters CL
- Number of aircrafts AC
- Number of versions VN

The generator places AP airports at random over the whole world using the usual coordinate system. On airport is special, it is the home airport. Only at the home airport maintenances can be done. It calculates the flight times between the airports as follows: A flight around half the world takes 24 hours. All other flights are shorter. The flight time is proportional to the distance between the airports. As next the aircrafts will be produced. The AC aircrafts will get one of the VN versions. The allocation is at random. The most complicated part is the generation of the clusters. Each aircraft will fly $\frac{CL}{AC}$ clusters. Every 30 days an aircraft has to fly to the home airport to do the maintenance. The first maintenance of each aircraft will be at any time during the first 30 days. The clusters are created as follows. The first cluster defines the starting airport; it is a flight from the home airport to this airport. The following flight uses this airport as the departure airport and arrives at any of the other airports. The next flight uses the last airport as the new departure airport and so on. This guarantees that there is at least one solution. The time between arrival and departure called ground time is randomly distributed over a fix interval. These procedure will be done for each aircraft separately.

The generated solution was used by the optimization algorithms described in Chap. 5 as the initial solution. It is interesting that the algorithms which search an initial solution described in Chap. 4 always found another solution than the one which was generated.

Appendix B

Data Viewer

To visualize data sets from our generator or from Lufthansa, we implemented a viewer. It can call the routines for initial solutions and optimization, as well as saving and loading calculated solutions. Clusters are visualized as boxes in different colors, for the different states of a cluster (arrived, deleted, maintenance event). It was very helpful to see the arrangement of clusters to fix bugs in our algorithms. In Fig. B.1 one can also see the reason why we could not use the data from Lufthansa: Clusters are overlapping.

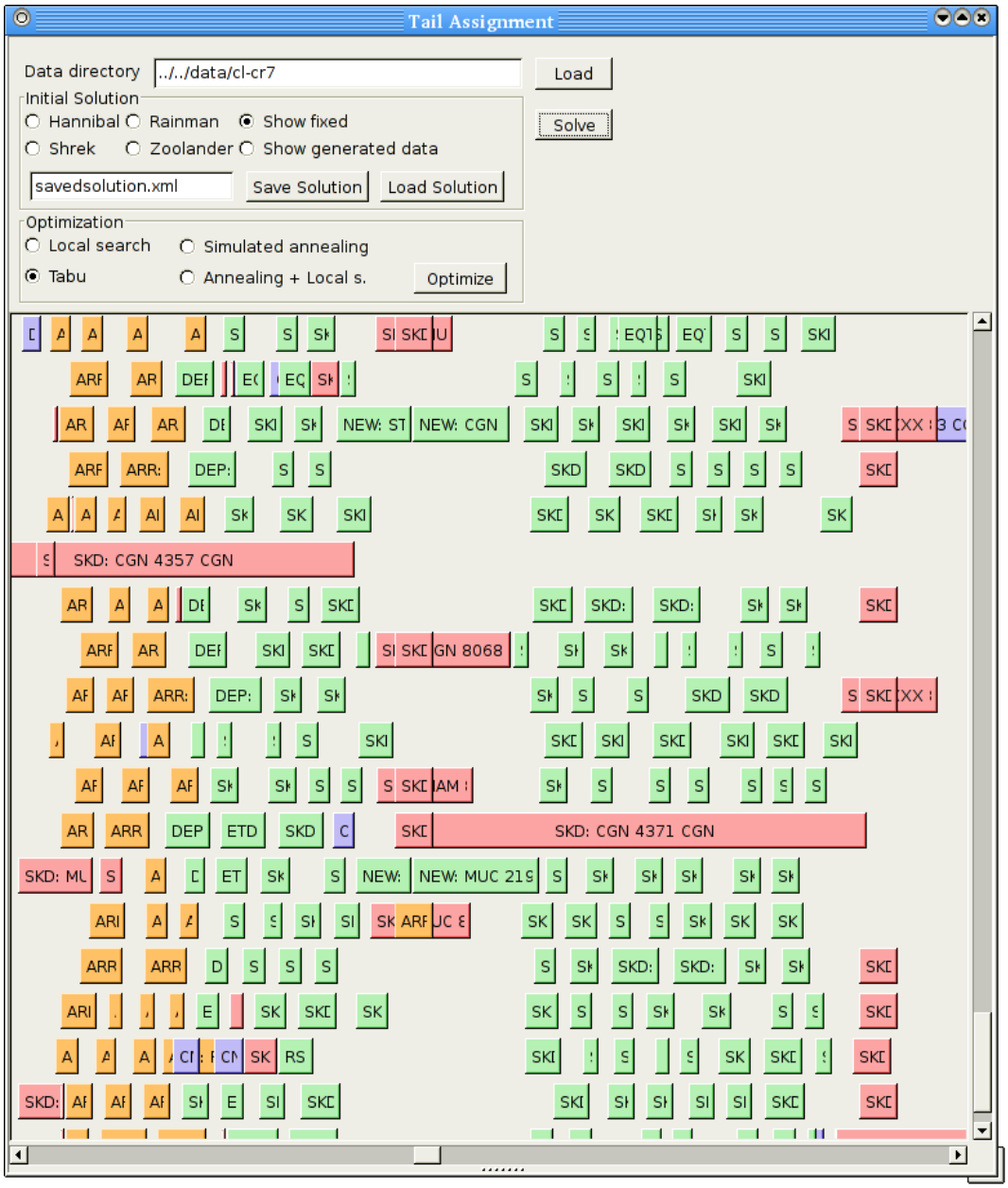

Figure B.1: Our data viewer, showing data from Lufthansa. The horizontal axis is the time, and on the vertical axis the aircrafts.

Bibliography

[1] B. Monien, T. Fahle, S. Götz, S. Grothklags, G. Kliewer, and M. Sellmann. Flugplanung mit Informatik Methoden (in german). *ForschungsForum Paderborn*, 2001.

[2] C. Barnhart, N.L. Boland, L. Clarke, E.L. Johnson, G.L. Nemhauser, and R.G. Shenoi. Flight string models for aircraft fleeting and routing. *Transportation Science*, 32(3):208–220, August 1998.

[3] R. Barták. *Constraint Programming: In Pursuit of the Holy Grail*, pages 555–564. MatFyzPress, June 1999. http://kti.ms.mff.cuni.cz/~bartak/downloads/WDS99.pdf.

[4] H. Alt, N. Blum, K. Mehlhorn, and M. Paul. Computing a maximum cardinality matching in a bipartite graph in time $O(n^{1.5}(m/\log n)^{0.5})$. *Information Processing Letters*, 37(4):237–240, 1991.

[5] D.T. Pham and D. Karaboga. *Intelligent Optimisation Techniques*. Springer-Verlag, London, 2000.

[6] O.C. Martin and S.W. Otto. Combining simulated annealing with local search heuristics. *Annals of Operations Research*, 63:57–75, 1996.

[7] C. Schaad N. Piotrowski and T. Schickinger. Optimization model for the NetLine/Ops optimizers. *Concept-paper, Lufthansa Systems Berlin GmbH*, 2004.